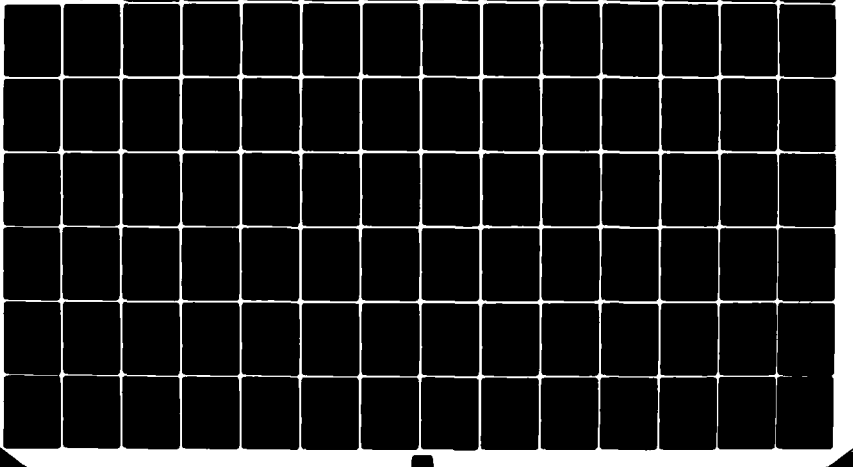
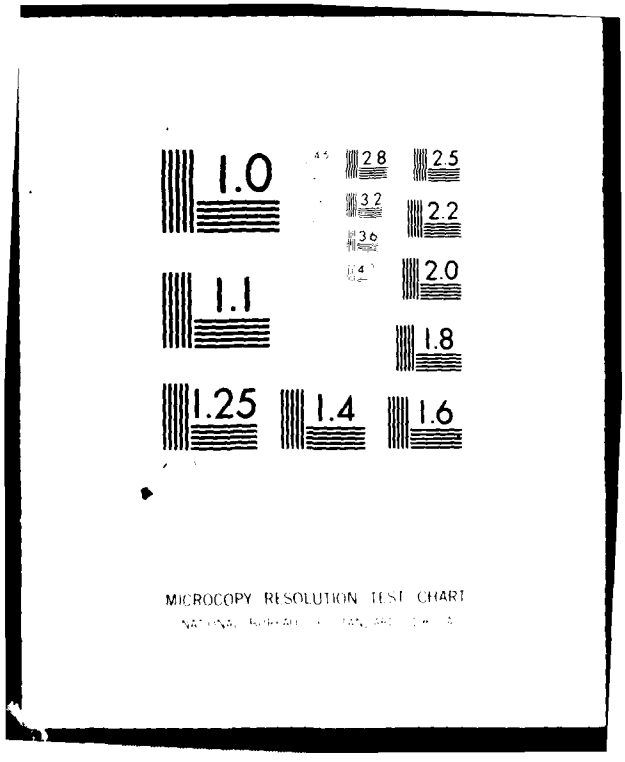
AD-A088 631 ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 9/2
GUIDANCE AND CONTROL SOFTWARE,(U)
MAY 80 A O WARD, P F ELZER, H G STUEBING
UNCLASSIFIED AGARD-AG-258 NL

1 OF 3
AD
AD-A088631




1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

AGARD-AG-258

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARDograph No. 258

Guidance and Control Software

NORTH ATLANTIC TREATY ORGANIZATION

DISTRIBUTION AND AVAILABILITY
ON BACK COVER

AGARD-AG-258

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARDograph No.258

GUIDANCE AND CONTROL SOFTWARE

Edited by

L.J. Urban
Technical Director and Deputy for Avionics Control (AX)
Aeronautical Systems Division
Wright-Patterson Air Force Base
Dayton, OH 45433
USA

This AGARDograph was prepared at the request of the Guidance and Control Panel of AGARD.

THE MISSION OF AGARD

The mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

- Exchanging of scientific and technical information;
- Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;
- Improving the co-operation among member nations in aerospace research and development;
- Providing scientific and technical advice and assistance to the North Atlantic Military Committee in the field of aerospace research and development;
- Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field;
- Providing assistance to member nations for the purpose of increasing their scientific and technical potential;
- Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

Published May 1980

ISBN 92-835-0267-1

Printed by Technical Editing and Reproduction Ltd
Harford House, 7–9 Charlotte St, London, W1P 1HD

PREFACE

The development of Computer Programs, which are referred to as Software, is currently on the critical path of all NATO weapon systems and developments. The cost of designing, developing, and subsequently maintaining software is currently costing many times the cost of the related hardware. The objective of this AGARDograph, assembled by the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development.

The AGARDograph is organized into two major parts. Part I deals with software design and management, while Part II covers software applications. We have been truly fortunate to obtain contributions in this AGARDograph from the most knowledgeable, experienced software experts on both sides of the Atlantic. Particular recognition must be given to the extensive efforts of the authors for developing the chapters which bear their names. We are grateful for the time these knowledgeable experts have taken to share their valuable experiences and lessons learned.

The Editor also wishes to express his appreciation to his panel colleagues Monsier Daniel Pichoud (France), Dr Reiner Onken (Germany), Mr Peter Kant (Netherlands), Mr John Hillington (United Kingdom), Mr Ronald Vaughan (US Navy), and Dr Herman Redeiss (US-NASA) who assisted me in the selection of topics and the identification of authors from their respective countries. I also wish to acknowledge the valuable contribution and assistance provided by the AGARD staff.

Special recognition is due my secretary, Mrs Agnes Vislosky, who handled most of my correspondence, assembled the final manuscript and even typed two of the last chapters submitted to enable their inclusion in this AGARDograph.

LOUIS J.URBAN
Technical Director
Deputy for Avionics Control
Aeronautical Systems Division
Wright-Patterson AFB, Ohio

CONTENTS

AN APPROACH TO THE DERIVATION AND VALIDATION OF REQUIREMENTS

by

A. O. Ward
British Aerospace
Aircraft Group
Warton Division
Preston PR4 1AX
United Kingdom

SUMMARY

The preparation of requirements is seen to be a relevant area to address in order to improve system software acquisition. There is evidence to suggest that it is potentially cost effective to consider means of improving the way in which requirements are validated. In order to validate requirements the form of expression and methodology of derivation need to be structured in a particular way. To achieve this a convenient interface with the engineer is required as well as an information structure amenable to validation and automated aids to this process.

Such an approach is described, embodying a hierarchial diagrammatic notation, information structure and an appropriate system description language and analyser to assist validation.

INTRODUCTION

This paper will discuss the problems associated with the development of requirements for Guidance and Control Software. It will suggest ideas that it is believed will overcome these problems and describe a specific application of those ideas in terms of a methodology and associated tools.

Let us first examine the position and contribution of requirements definition in the perspective of the overall product life cycle. In doing so we will consider it from two relevant viewpoints.

- The resources needed to produce requirements.
- The quality of requirements and its impact on budget.

A typical phased life cycle is shown in Fig. (1) as consisting of a number of logical steps providing both technical and management outputs. There is general agreement that phasing product development in this way is desirable although there is still much discussion as to the specific interfaces between phases and the methodologies and tools used to achieve them.

The following observations can be made:

The requirements phase represents a small percentage of overall budget and is labour intensive. A typical budget profile taken from (1) is shown in Fig. (2). The cost of changing software either to correct errors or because of modified requirements becomes more expensive as the life cycle proceeds. This is exemplified by Fig. (3), based on Boehm (2), where the cost can be seen to be increasing by orders of magnitude between requirements and operation. Development apart, requirements are less likely to change if they are considered in depth to begin with and quite clearly some errors originate from a poor communication interface between customer and supplier.

These observations lead to the conclusion that benefits are to be gained by improving the way in which requirements are developed. A large percentage increase in resources applied to the requirements phase will result in a small change to the overall budget and hence improving the production of requirements has a potentially advantageous cost leverage. Having established a case for examining the production of requirements in order to ascertain how it can be improved we will discuss below the nature of the problems encountered in more detail. Without wishing to pre-empt this discussion it is believed that these problems can be overcome by the following:

- Enter the production of requirements by considering as many technical viewpoints as possible.
- Provide the engineer with a convenient notation for expression.
- Proceed via decomposition to produce a hierarchy of functions and data.
- Establish an information structure that enables validation.

- Enlist the help of tools that will assist validation and simplify the production of documentation.

In this paper we will discuss the roles of a standard for the expression of requirements, a methodology to be used in their derivation and tools to assist us. A particular set will be described that currently appear to be adequate while accepting the necessity for their development in the light of experience.

PROBLEMS ENCOUNTERED IN THE PRODUCTION OF REQUIREMENTS

There is a growing feeling on both sides of the Atlantic that many of the problems associated with software projects can be traced back to inadequacies in the requirements specification phase. This observation is important when one considers the cost of correcting problems over the product life cycle. As mentioned above, Boehm has shown that there is a significant difference between the cost of correcting errors during requirements specification and when the product has reached the testing phase.

Unfortunately, as Rubey so effectively points out in (3), the organisation of software testing often means that the most significant errors are detected late in the testing activity. The initial unit testing usually only detects the errors made by the programmers when the unit under test was programmed. The integration testing generally only detects errors that result from an incomplete or ambiguous software design. Acceptance testing, done last, detects errors that were made very early on when the functions that the programme satisfies were defined. Clearly the early verification of the original functional requirements is essential.

In TRW's software reliability study (4) the analysis of error data showed that most of the errors were design and requirement errors as opposed to coding errors and those made during the correction of other problems. The evidence indicates that although software development projects typically expend much effort in requirements and design reviews, these sources of error were shown to represent major portions (60%) of the total errors detected during formal testing.

Reifer has reported more recently in (5) the results of project experience indicating that more than 62.5% of all changes during test and integration resulted from latent requirements and design errors. In turn about 85% of those errors were a result of inadequacies or deficiencies in requirements.

Perhaps the most comprehensive collection of qualititive evidence based on project experience is to be found in the MITRE study (6) for the United States Department of Defense. This work was aimed at translating software acquisition and management problems into specific objectives that the DoD could then pursue as part of its research and development programme. The comments in the project interviews that relate to requirements specification are quoted below.

- Requirements were not well developed or understood by the user and the sub-contractor.

- Requirements change during development; programme managers should try to get user agreements early in the process.

- Some of the major software problems and delays were caused from software not performing the functions that management intended.

- It takes too long to develop, produce and deploy weapons systems (up to 10 years). Requirements change over the development cycle which causes expensive redesign efforts.

- It is important to validate interfaces early.

- There is a need to stabilise requirements early in the system acquisition. Changes to the requirements caused major software redesign efforts which delayed the operational date of this system. With extended development of a system there is a danger it will be overtaken by the technology changes.

- There is a need to do a good system analysis of requirements, define all interfaces and to ensure that all components of a system can be built before starting the detailed software design and coding.

- There are at times software compromises made at the beginning of a system development because of the lack of funds. However, there always seems to be funds to correct for software deficiencies later.

- There is a need for a research and development programme to find ways to accomplish automatic verification through simulation or other means to avoid expensive testing.

- Changing requirements presented major problems; trying to cope with changes was expensive and time consuming.

- Functional requirements should be clearly stated before entering into a development contract. A good check on feasibility of requirements is to determine whether they are testable, if not they should be eliminated.

The quotations were given in full in order to emphasise the repetition of certain observations over the many projects examined.

- Requirements should be validated at an early stage.

- Changes to requirements are the norm and have expensive consequences.

- There is a communication problem between the customer (requirements) and the contractor (product) that leads to lack of conformance.

- Requirements need assessing to ensure that they are practicable.

We can conclude therefore that in order to improve the situation we require standards and tools that will satisfy the following objectives.

- Validate requirements (i.e. ascertain if they are consistent and complete).

- Assess the consequence and potential cost of changes to requirements.

- Improve the communication between the customer and contractor (i.e. unambiguous).

- Provide the detail that will allow realistic assessment of whether requirements are practicable.

It could also be said that another cause of changes to requirements is that they were inadequately considered originally and so any improvement in their production should include mechanisms that assist the engineer in making his statement as comprehensive as possible. This can only be done by allowing individual viewpoints to be expressed and subsequently combined into an overall system requirement.

This latter point is important when considering new generations of avionic systems, moving away from centralised computing and equipment oriented subsystems towards a fully integrated approach with distributed and even federated processing. The latter approach, to be effective, should as far as possible have no preconceived ideas concerning equipment boundaries before functional requirements have been prepared.

It can be concluded from the above that the quantitive evidence justifies the case for improving the way in which we produce requirements through leverage on the overall budget. The qualitative evidence points to specific ways in which the process could be improved, such as techniques for validation and improving customer/vendor communication.

AREAS OF IMPROVEMENT

This section will discuss how the problems of communication, validation, conformance and the consequence of change can be addressed. The solutions are intrinsically related to the structure of the requirement both to improve communciation and assist validation although differing in the resolution of detail.

Requirements are presented in varying levels of detail as a starting point.

The customer may have just a notional idea of what he requires of his system and will rely on the implementor to produce a more detailed statement in partnership with him. At the other end of the scale, the customer, because of his experience may be able to transfer his notional requirements into the detail needed without assistance.

With either approach it is essential that the detailed end product conforms with the original (or notional) requirement. Notional, perhaps is too trivial an adjective but it is attempting to show that the customer views the system requirement, initially, as part of a global requirement at a higher level. For example, as part of a strategic scenario a front line ground support vehicle with reconnaissance may be required, to which can be assigned certain qualities such as navigation performance, weapon aiming accuracy etc. This high level requirement must then be translated into a system requirement which can be seen to be part of an overall vehicle requirement. At a lower level, the system requirement is itself seen to consist of specifications for particular pieces of hardware and matching software requirements.

Clearly the transformation and enhancement of the original requirement down to the detail of a software requirement which can act as the basis for software design is a long and laborious process. It is also vital that during this process conformance between the starting point and the end product is preserved and that nowhere along the way inconsistencies and ambiguities accumulate that will lead to erroneous requirements or requirements that will confuse their recipient the software designer.

It is believed that this can be achieved by developing requirements within a hierarchial framework, where each level in the hierarchy represents a more detailed qualification of the problem. Ideally the hierarchy should be developed from the overall system requirements and transcend into the software requirements but at the very least be employed for software requirements.

Thus, in a process hierarchy approach a system is organised as a sequence of hierarchial levels of processes. In each level a group of interacting processes can be observed which is accomplished by yet another group at a lower level (7). This general approach to describing systems is varyingly described as top down, structured, stepwise refinement and functional decomposition. Although there are subtle differences in these techniques, in practice, they subscribe to the philosophy outlined above.

This approach has been the basis of successful engineering practice for decades. A requirement is initially satisfied by a conceptual or outline solution that is gradually more detailed until an adequate description of the system results that will allow it to be built. This process allows iterations to take place between the various levels.

Fig. (4) is a simplified representation of the drawing scheme used to manufacture an aircraft and it is possible to make the following observations.

- There are several levels of detail.
- At each level of detail the customer and designer assess in turn whether the design is practicable, will satisfy the requirements and if it is correct.
- The hierarchy of information that each drawing level represents can be seen to be a logical decomposition of the preceding levels.
- There is an unambiguous method of expressing the design (i.e. a drawing system with standards).
- Inter-relationships between various levels and drawings at the same level are referenced on the diagrams.

In short, there is a visible structure of the information required to construct the aircraft.

When applied to the development of system and software requirements it is clear that such an approach if applied rigorously would enable conformance to be established via a series of small increments of detail. Equally the effect of changes to the requirement could be quickly traced through the hierarchy in order to establish the functions affected by such a change. The notation in the engineering analogy allows an unambiguous statement of the requirement which is vetted against a drawing standard. In the case of system/ software requirements a standard is needed that imposes an information structure on the description that prevents ambiguities and inconsistencies.

As a very simple example consider a small subset of the information categories required in describing process and data.

PROCESS

Name:	A recognisable and unique identifier for the process.
Part of:	It should be seen to be part of a process at the previous level.
Parts are:	The names of processes that are part of this process at the next level.
Uses:	The data used by this process.

DATA

Name:	A recognisable and unique identifier for the data.
Part of:	It should be seen to be part of data at the previous level.
Parts are:	The names of data that are part of this data at the next level.
Used by:	The names of processes that use this data.
Derived by:	The name of the process that derives this data.

The validation of information provided in these categories falls into two areas:

- Whether the information is present or not and its consequence.
- If the information is present, is it consistent with corresponding information in other categories and relating to other parts of the system.

The scale of the validation tasks associated with these two classes differ by an order of magnitude. The complexity of the latter task is further magnified when one considers that the categories described above represent a very small percentage of all the possible categories of information that would be expected by a comprehensive requirements standard. The function of the standard is that it has made validation of the requirements possible (i.e. the validation procedures are now visible). The function of an automated aid is that it makes the validation process practicable by alleviating the considerable clerical task of checking the many categories of information necessary.

A SPECIFIC APPROACH TO THE PRODUCTION OF REQUIREMENTS

The ideas discussed in the previous section have been used to devise a specific approach to developing systems and software requirements. This work is aimed at deriving the requirements for a fully integrated avionic system of the type being considered for current and future military aircraft. The techniques and tools are collectively described as Semi Automated Functional Requirements Analysis (SAFRA). The major elements of SAFRA are shown in Fig. (5) and can be seen to contain the following:

A Method of Problem Entry and Decomposition

This consists of bounding the problem to be described and in doing so proposing the viewpoints to be considered while developing the requirement. Information is collected for each viewpoint including its impact on other viewpoints and represented in tabular form. The tabular data are decomposed and used to identify threads (or logical paths of operation) through the actions required to satisfy the input and output data. The threads are reconciled to produce combined threads both within viewpoints and across viewpoints where iteration between them occurs.

A level of decomposition is defined as a statement of a new set of viewpoints reflecting the changing influences brought about by increasing the level of detail.

A Notation for Description

The requirement at the highest and subsequent levels is represented using a simple diagrammatic notation which allows decomposition of data and process to proceed in parallel. All the elements contained within the diagrams are described via lower level diagrams and associated text referred to as object definitions.

An Information Structure for Validation

The information contained in the object definition is translated into the more rigid format of a Technical Definition for each object and for a number of object types. The format consists of particular information categories which are in part of a general nature or specific to the object type.

Validation

The validation procedure is used to check the consistency, completeness and ambiguity of all the information categories in a Technical Definition. These tests validate hierarchial statements and also cross references to related Technical Definitions.

Practically all of these tests can be accomplished using a system description language with associated analyser and data base.

In the context of SAFRA the particular methodologies and tools employed are as follows. The method of problem entry and notation are based upon the Controlled Requirements Expression (CORE) methodology developed by System Designers Ltd., in the United Kingdom. The information structure to assist validation also owes much to System Designers' standards for requirement specification.

The automated aid used for validation is the University of Michigan's Information System Design and Optimisation System (ISDOS) consisting of a Problem Statement Language (PSL) and Problem Statement Analyser (PSA). The above features of SAFRA will now be discussed in a little more detail.

Problem Entry and Decomposition

The top level of a requirement should be achieved by considering as many viewpoints as possible that relate to external interaction with (or influence on) the system being described, (e.g. tactical user, pilot etc.). For a particular viewpoint information is assimilated by the engineer directly or by interviewing a relevant specialist using a checklist of questions. Experience to date indicates that this list begins as an aide memoire but once the engineer becomes more practiced it can eventually be dispensed with. A typical question checklist is shown in table (1).

This collecting of information is the first of ten logical steps taken at each level of decomposition. Subsequent steps relate to data reconciliation, data decomposition and the construction of isolated and combined threads through each viewpoint and across viewpoints. The threads are represented by the diagrammatic notation described below. In addition to threads an operational (a 'snapshot') view is considered in order to describe the system in operation, indicating, for example, the degree of parallelism required.

The final step is a 'reliability' assessment, where every object (data and process) is examined in order to ascertain the consequence of its failure or degradation on the rest of the system. This check will result in system changes if corrective or recovery action is required.

The reader is referred to reference (8) for a more comprehensive discussion of the methodology underlying Controlled Requirements Expression (CORE).

1. Establish system title.
2. Define system purpose (from previous level).
3. Define system boundaries (for the particular level).
4. Establish Decomposition Viewpoint(s).
5. Define Decomposition Level.
6. ACTIONS

   What Actions does the ..... system perform?
   (How are these Actions performed?) (By what mechanism)
   Prompts: Operator/Computer/Transmitter/Receiver
   Are there any other Actions?
   (How are these Actions performed?)
   Are you sure thats all?
7. What are the inputs for the Actions above?
8. What are the sources for these inputs?
9. What are the outputs for the Actions above?
10. What are the destinations of these outputs?
11. What Events occur that start or stop these Actions?
12. What Action does the ..... system perform to Gather Information?
    Prompt: Input signal transducers
    (By what mechanism does the ..... system Gather Information?)
    Prompt: Sensor/Operator
13. What Action(s) does the ..... system carry out on Inputs?
    Prompt: Signal Processing/Computing
    (By what mechanisms are these Actions performed?)
    Prompt: Aircraft systems/operators/processing
14. What Information Checking does the ..... system perform?
    Prompt: Signal validation/Signal correlation
    (By what mechanism is this change conducted?)
    Prompt: Computing : hardware/software
15. What Information Storage does the ..... system perform?
    Prompt: Mission Data Store (e.g. Target position/Waypoint (Fuel))
    (By what mechanism is this storage achieved?)
    Prompt: Alterable store/permanent store
16. What Dissemination Action(s) does the ..... system perform on the output data?
    Prompt: Transducing/Output Data processing
    (By what mechanism(s) is this dissemination achieved?)
    Prompt: Data Formatter/Transmitter
17. What Correction/Change Action does the ..... system perform?
    Prompt: System updates/course corrections
    (By what mechanism(s) are these Corrections/Changes achieved?)
    Prompt: Operator/Aircraft Systems.
18. What management/control action does the ..... system perform?
    Prompt: Engine control - by the Main Engine Control Units
            Flight control - by the C.S.A.S.
    (By what mechanism(s) is management/control achieved?)
    Prompt: Operator/Computer
19. What Failure Detection; Location and Diagnostic Action; does the ..... system perform?
    Prompt: Internal Testing
    (By what mechanism(s) are failures detected; located; and diagnosed?)
    Prompt: Test circuitry
20. What are the Inputs for the Actions in 12 to 19 above?
    (By what media are these Data Transmitted?)
21. Is that all?
22. From what sources are each of these inputs derived?
23. Is that all?
24. What are the Outputs for the Actions in 12 to 19 above?
    (By what media are these Data Transmitted?)
25. Is that all?
26. What destinations are these outputs passed to?
27. Is that all?
28. What Events occur that start or stop the system?
    (When do these Events occur?)
29. For Decomposition Questions 6 to 14 apply.

TABLE (1)
TYPICAL QUESTION CHECKLIST

Diagrammatic Notation

The basic technique of description is a hierarchic set of diagrams. An item which appears on one diagram is itself decomposed and described in a lower level diagram. Each diagram is supported by notes and the whole is supplemented by a complete Technical Definition which always contains information under key headings, which we will discuss later. This section will attempt to describe the notation only.

The system adopted allows strict control of the consistency of one level of decomposition with the higher level from which it was derived and is based upon boxes and directed lines. On each diagram a box defines an object and a line defines its interface, interaction or area of external effect. The interfaces to a box at one level must appear as external interfaces in the lower level decomposition description.

As most systems contain two fundamental hierarchies (process and data) the production of a description aims to represent both hierarchies via a simple notation. This is done using two sets of diagrams:

- Data diagrams
- Process diagrams

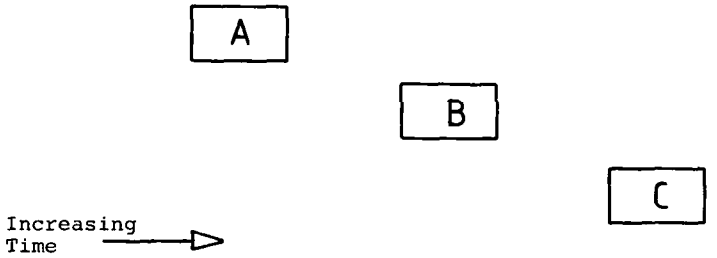
In a data diagram a box represents information and a directed line normally represents a process. In a process diagram a box represents a process and a directed line represents information.

The side of the box at which a directed line appears indicates the type of relationship between the line and the box.

The difference between these two approaches in practice is shown in Fig. (6) and can be termed 'data on arrow' and 'process on arrow'.

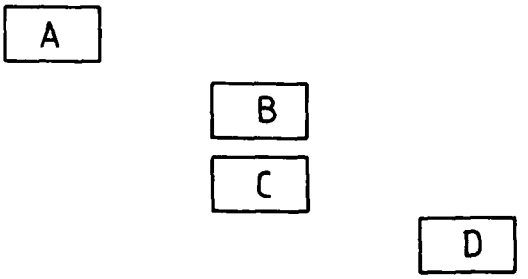
Box Ordering

Temporal ordering of data production or process takes place from left to right, within a particular diagram.



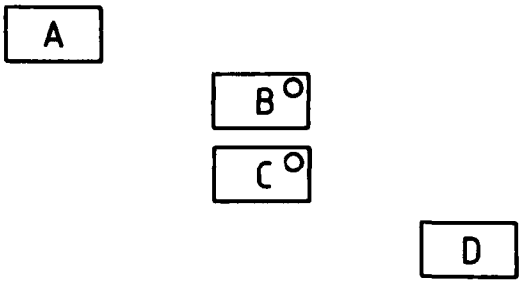
If B were a process this means that it can take place at anytime between the end of A and the start of C. If B were data, it is produced between the time A has been produced and the time C starts being produced.

Intermediate order is shown vertically.



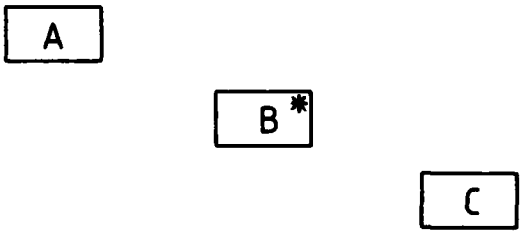
B and C may occur in any order, including in parallel, but both must occur after A and before D.

Mutual exclusion is indicated by a O in one of the top corners of a box.



Either B or C, but not both, will occur after A and before D for process diagrams, which occurs will depend upon the selection made by A.

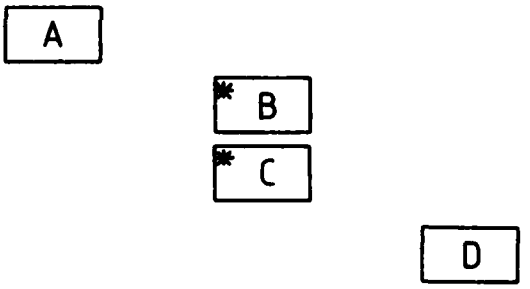
Iteration is indicated by an * in one of the top corners of a box.



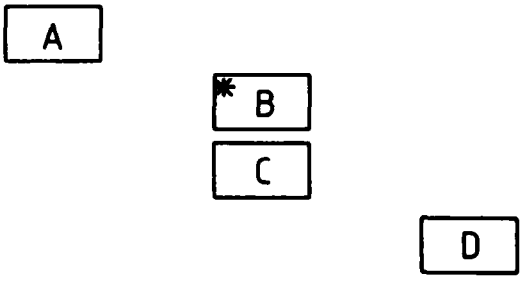
B occurs repeatedly after A and before C, this may include zero occurences.

Combined Ordering Notations

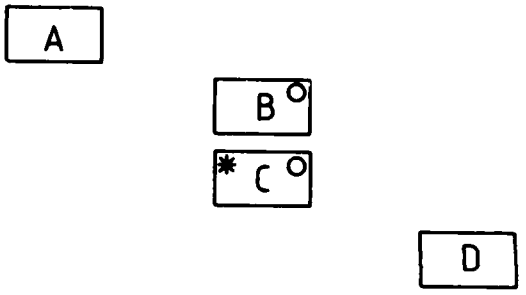
The above notation may be combined in such a way as to represent the following cases:



Here B and C occur repeatedly after A and before D, and instances of B and C can occur in any order or in parallel (e.g. asynchronous process).



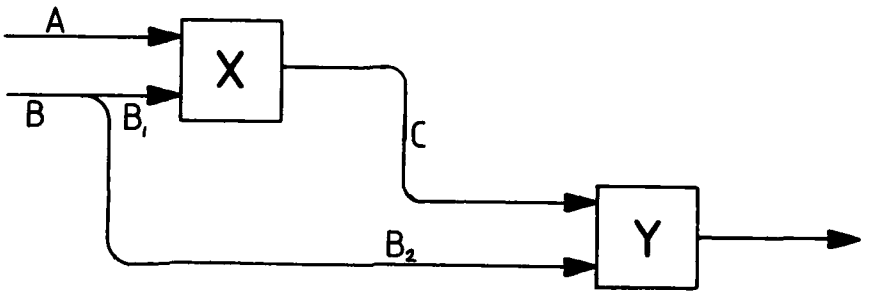
Here, B occurs repeatedly and C occurs once, i.e. after A and before D. Each instance of B can occur before, after or in parallel with the one instance of C.



Here, either one instance of B or several instances of C occur after A and before D.

Line Branching

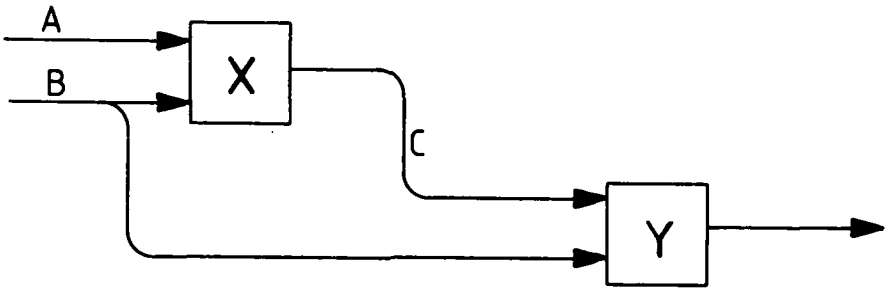
Lines interconnecting boxes may be branched appropriately to indicate either decomposition or duplicate use of whatever the line is representing.



Here B is a composite of B1 and B2

For process on arrow: X is derived partly by A and B1;
Y is derived partly by C and B2;
Parts of B derive parts of X and Y;

For data on arrow: Part of X uses A and part uses B1;
Part of Y uses C and part uses B2;
Parts of B are used by part of X and part of Y;

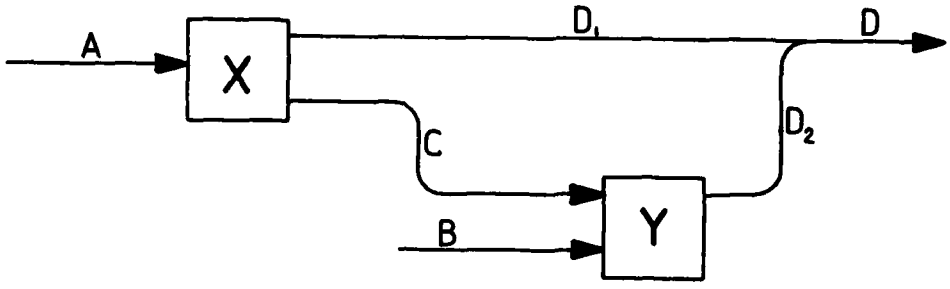


For process on arrow: X is derived partly by A and B;
Y is derived partly by C and B;
B derives parts of X and Y;

For data on arrow: All of B is used by part of X and by part of Y;
B is not being decomposed.

Line Joining

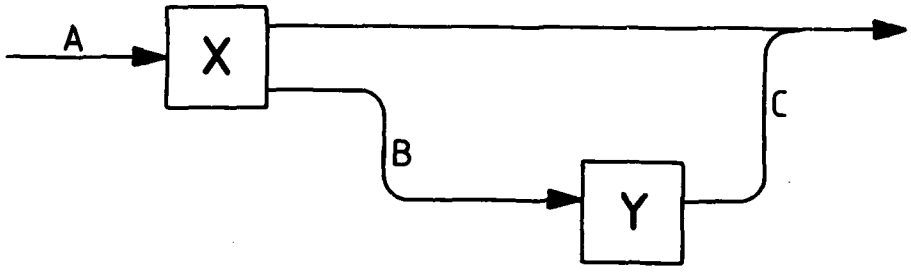
Similar rules apply to line joining.



Here D is a composite of D1 and D2

For process on arrow: Part of X is used by D1 and part by C;
Part of Y is used by D2;
Parts of X and Y are used by parts of D;

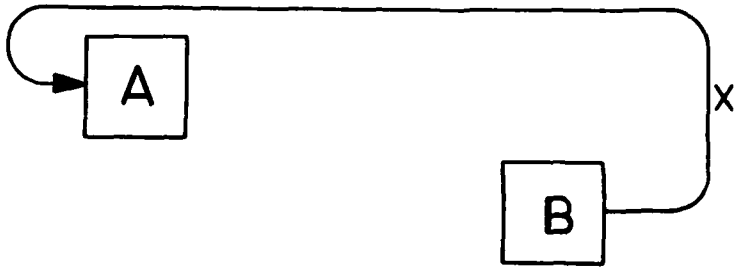
For data on arrow: Part of X derives D1 and part derives C;
Part of Y derives D2;
Parts of X and Y derive the parts of D;



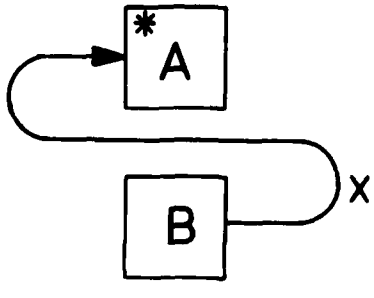
For process on arrow: Part of X and all of Y is used by C;

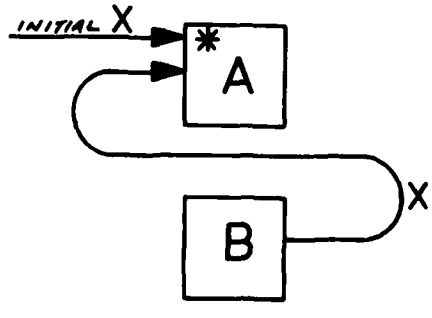
For data on arrow: All of C is derived by part of X and by part of Y;
C is not being decomposed;

A final word about lines and boxes.



The data X cannot travel backwards in time. This construction is possible only if A and B are parts of the same iteration box.





X can travel backwards only if an iteration at a higher level contains A and B or if A also receives a default (initialisation) value.

Object Definitions

The diagrams above are insufficient and in order to provide a complete system description additional information is required. The nature of this information is such that it would be inconvenient to include it in the diagrams and therefore it is provided as additional text. The function of the object definition is to record all the information required for system description prior to it being transferred into the more formal representation required for subsequent validation. It consists of a data or process diagram with an appropriate indexing key and associated text.

In compiling a description a certain level of confidence in its accuracy and completeness must have been reached. In practice this is achieved by addressing the levels below before judging the description at the current level. Thus at least two levels below the current one will have been explained in some detail before the object definition of the level under consideration is seen to be adequate.

A typical diagram proforma is shown in Fig. (7) with appropriate key and should be accompanied by change and review sheets and notes providing technical information not provided by the diagrammatic notation. For example, the following additional information would be required in defining a process.

- A brief description of the process
- The maximum time allowed for this process
- The process at a lower level that makes up this process
- A process at a lower level utilised by this process
- A process at a higher level that utilises this process
- The frequency of this process (if iterative)
- Security classification of the subject being described
- The event that triggers this process
- The event triggered when this process is terminated

Technical Definitions

General

The function of a Technical Definition (T.D.) is to represent the system description in such a way as to allow validation of the requirement. This is achieved by partitioning the description into specific and unambiguous information categories, grouped according to the type of object being described.

The categories apertain to hierarchies, relationships, qualities, time ordering etc., of the object they are helping to describe. The titles assigned to these categories correspond to PSL constructs and such words are identified in the T.D. description given below, by being in capitals.

The objects used in system description are:

PROCESS

INPUT

OUTPUT

SET

GROUP

ELEMENT

INTERFACE

Brief Definitions of the Objects are:

PROCESS: Represents any action carried out on data. It can validate Inputs, produce Outputs, store and manipulate data to meet the objectives of the system and cause the initiation of additional Processes.

INPUT: Describes a collection of data produced external to the system but used by the system. It shows the flow of data from the outside world into the system.

OUTPUT: Describes a collection of data produced by the system, but is used external to the system.

SET: Defined as a typical or logical view of the data as seen by the user. It is a collection of one or more occurences of objects that contain or carry data values. In this context it is applied to the medium of input or output.

GROUP: A logical collection of data elements and/or other groups. In this context it is used to describe data generated internal to the system.

ELEMENT: The basic unit of data and therefore cannot be sub-divided. An element is used to describe a data object which may take on a value. In this context it is used to describe data generated internal to the system.

INTERFACE: An object or system outside the boundaries of the target system that interacts with the system being described.

In addition to the above, T.Ds for events and the conditions that control them have also been considered.

A Technical Definition for a Process is given below as an example.

PROCESS TECHNICAL DEFINITION

PROCESS:	The name of this PROCESS;
SYNONYM:	An appropriate synonym;
DESCRIPTION:	A short description of this PROCESS starting on this line and preferably not more than 5 lines in length, terminated with a;
KEYWORDS:	An appropriate identifier that can be used for selective retrieval from the database;
ATTRIBUTES ARE:	
TIME-LIMITS	The maximum time allowed for this PROCESS;
SEE-MEMO:	The name of the document that calls up the requirement for this PROCESS;
GENERATES:	The OUTPUTS generated by this PROCESS;
RECEIVES:	The INPUTS received by this PROCESS;
SUBPARTS ARE:	The PROCESSES at a lower level, that make up this PROCESS;
PART OF:	The PROCESS, at a higher level, that this PROCESS is part of;
UTILIZES:	A PROCESS at a lower level utilized by this PROCESS;
UTILIZED BY:	A PROCESS at a higher level that utilizes this PROCESS;
USES:	Internal data used by this PROCESS;
DERIVES:	Internal data derived by this PROCESS;
PROCEDURE: AFTER	The PROCESS that immediately precedes this PROCESS in time,
BEFORE	" " " " follows " " " "
HAPPENS:	Frequency TIMES-PER: Interval; (e.g. HAPPENS: 30 TIMES-PER second)
TRIGGERED BY:	The event that starts this PROCESS off;
INCEPTION-CAUSES:	Events generated by the inception of this PROCESS;
TERMINATION CAUSES:	The event generated when this PROCESS finishes;
RPD:	The engineer responsible for this definition (e.g. Responsible Problem Definer);
SECURITY:	The security classification of this Technical Definition;
SOURCE:	Information not contained within the system documentation;

VALIDATION

General

The Technical Definitions described in the previous section contain information that may be examined as part of a validation process. Validation consists of examining the requirement in order to ensure it is consistent, complete and unambiguous.

Completeness, here, refers to the completeness of a stated system structure and not whether all the technical viewpoints have been accommodated. The latter will only emerge from technical reviews of requirements.

In this section we will describe in detail a typical validation procedure, as applied to a Process Technical Definition, in order to demonstrate the nature of the tests being made and the files required to support them. In the next section we will summarise how PSL/PSA assists this procedure but it is important that we clarify the specific tasks that the tool will undertake by first considering a manual approach to the problem.

Support Files

Before we discuss the tests there are a number of files that need to be established and these are as follows:

System Dictionary

This contains the names of all objects referred to in a system description. This includes items mentioned within Technical Definitions as well as the objects for which T.Ds already exist. i.e. Object, synonym, type (e.g. Read Data, RD, Process).

Technical Definition File

This contains the names of all objects that have been qualified by a Technical Definition. As such, all objects referred to at levels of decomposition higher than the one being considered should have an entry in this file. Similarly, all objects being qualified at the current level should be entered in this file. The file is structured according to type (i.e. Process, Input, Output etc.).

Responsible Problem Definer File

This contains the names of all personnel involved in the definition of the requirement, with details of the specific objects they are responsible for.

Memo File

This contains the identifiers of all ancilliary documentation referenced in the system description with details of specific objects that reference them.

Validation Tests

Validation takes place once Technical Definitions have been prepared for a level of decomposition. In brief it consists of checking each Information Category (I.C.) in the T.Ds for completeness, consistency and ambiguity.

For example:

- Completeness; has the I.C. been completed and if not is the description still valid?
- Consistency; are the statements made about the objects connected with this Technical Definition consistent with complementary statements made in other Definitions? (e.g. Inputs generated and Outputs received).
- Ambiguity; Have the names assigned to objects in one T.D. already been used in another context elsewhere?

Checklists of tests or statements of error can be drawn up to satisfy the above criteria and an example for the validation of a Process T.D. is shown in Fig. (8) .

Answering the questions with a YES constitutes a pass, NO, a failure.

The tests may be classified as follows:

- General comments
- General section
- Hierarchies

General Comments

All entries should be checked for completion and blank entries reported. Not all of these will constitute a failure, for example a Process is not necessarily utilised by another PROCESS. At the top and bottom levels of decomposition, the part of and subparts are entries respectively will not apply.

Hierarchies

These in turn can be seen to be relating to

- the level above
- the current level
- the level below

Level Above

All the identifiers for T.Ds at the current level will have been included in the description of the previous level. They will therefore appear in the System Dictionary. The identifiers cited at the current level must be checked against the System Dictionary to ensure that they exist and are being used in the correct context.

Statements of error would be:

Name not in System Dictionary

Name used for another purpose

Clearly, there must be one parent at the previous level, for the objects at the current level and a T.D. must exist for it.

e.g. More than one name supplied

Entry does not exist

Current Level

All identifiers referred to at the current level must have T.Ds prepared for them and hence entries in the T.D. file.

e.g. Entry does not exist

These T.Ds will contain complementary statements that relate to the T.D. being considered (i.e. a DERIVES category in a PROCESS T.D. will have a complementary DERIVED BY statement in a GROUP T.D.).

e.g. Cross References do not exist

Level Below

Names used for the level below should not already exist in the System Dictionary.

e.g. Name in System Dictionary

Clearly there should be more than one offspring, at the level below, of a parent at the current level.

e.g. Only one name supplied

These errors are summarised in the table below:

	LEVEL	SYSTEM DICTIONARY	TECHNICAL DEFINITION	ANOTHER PURPOSE	CROSS REFERENCES
ERROR STATES	ABOVE	NOT IN	NOT IN	YES	-
	CURRENT	NOT IN	NOT IN	YES	NO
	BELOW	IN	IN	YES	-

PSL/PSA ASSISTANCE

General

In this section we discuss the role of PSL/PSA in SAFRA and how it can be used by the engineer to validate and assess the completeness of his description.

In order to assist the engineer in preparing the Technical Definitions and to produce them in such a way that they can be converted into PSL, a number of proforma have been devised. An example is shown in Fig. (9).

Files

The files used to assist validation that were described in the previous section can be generated by selective use of PSA reports as follows;

System Dictionary

Every object name that successfully passes syntactical and semantic checks by the analyser will be stored in the database. It may be subsequently retrieved and displayed using the Dictionary Report. At minimum this will allow listing of all names stored in alphabetical order, with synonym and type. An example is shown in Fig. (10). The majority of names will have an implied type although they have not been explicitly assigned, (e.g. SUBPARTS ARE of a Process must be themselves Processes). In this case the type will be automatically assigned by the analyser. On the other hand the context of some objects are ambiguous, (e.g. A Group may CONSIST OF Groups or Elements). In this case the name will be stored in the database, but examination of the Dictionary will show its type as undefined.

Technical Definition File

A Technical Definition is very similar in structure and content to a PSA Section. A Section consists of all the valid PSL statements that relate to a particular object type. An example for Process is shown in Fig. (11). The Formatted Problem Statement report will list, for a particular object, all the statements entered into the database, as shown in Fig. (12) and can be used to represent a Technical Definition File.

Responsible Problem Definer and Memo File

These may be generated as above, by use of the Formatted Problem Statement report to list the Responsible Problem Definer and memo sections, as shown in Fig. (13) and (14).

Validation

The reader is referred to Ref. (9) for a complete description of the use of PSL/PSA in checking a system description, here we will discuss those particular aspects that relate to the validation procedures outlined in the previous section.

A considerable portion of the error detection facilities in PSA are used to check the preciseness of new PSL statements being added to the database. The analyser checks that the syntax is correct and that the user-defined names given in the new statements are consistent with the names already in the database. If either of these conditions fail, an error diagnostic is generated by the analyser to inform the user that the information to be stored in the database was ambiguous or inconsistent. No ambiguous or inconsistent information is allowed into the database.

It is important that once an object is defined and has an associated object type (e.g. Process or Set), the object can only be used in the context in which it is defined. Thus an object defined to be a Process cannot be also used to define a Group of data.

Similarly, only valid relationships between objects are allowed. For example, a USES relationship between two Process names is not permissible and any attempt to specify this would generate an error diagnostic.

Let us now examine the validation tests for hierarchies, described in the previous section, in order to identify how PSL/PSA provides assistance.

Level Above

Name not in System Dictionary: the name will have been automatically entered at the previous level description and if not, the reasons for omission would have been reported via an error diagnostic.

Name used for another purpose: this will be reported by the analyser when a new block of PSL is input (i.e. 'Name already used in different context').

More than one name supplied: this will be reported by the analyser when a new block of PSL is input (i.e. 'Already part of something else'). This only applies to system structure relationships (i.e. Part of/Subparts are), data structure relationships (i.e. Consists of/Contained in) are such that objects may be Contained In several objects at a higher level.

Entry does not exist: The Formatted Problem Statement report would be examined.

Current Level

Entry does not exist: The Formatted Problem Statement report would be examined.

Cross References do not exist: In most cases PSA makes appropriate cross references by inserting complementary statements in the relevant sections. For example, if an interface T.D. Generates an Input, the Input T.D. will have a Generated By statement inserted referring to the relevant Interface.

Level Below

Name in System Dictionary: Manual Inspection of Dictionary report. This is particularly important as entering a name that already exists in the system will merely reopen the associated section and merge the new statements with those already in the database.

CURRENT STATUS OF SAFRA

An approach to the development and validation of requirements has been described that it is hoped will overcome the problems described in the earlier sections. SAFRA is currently at the preliminary stage but development of its constituent elements is proceeding along a broad front, addressing in-particular

- Detailed application and administration of the methodology of decomposition (CORE).
- Assessing the impact of an automated aid on Configuration and Quality Control Procedures.
- Extending the information categories within Technical Definitions (and hence the scope of PSL) to cater for all the elements of system description encountered in practice.

SAFRA is being applied in two contexts:

- As an exploratory exercise it is being used to develop the requirements for an integrated avionic system from the highest level.
- In a more specific project it is being used to produce the requirements for a particular subsystem alongside more conventional approaches to the problem in order to assess its relative value.

REFERENCES

1. Tools for Software Development; E. F. Miller, European Computing Review, March 1979
2. Software Reliability; Measurement and Management; Dr. B. W. Boehm, 2nd International Software Management Conference, Nov 1977
3. Verification of Requirements and Design through Structured Analysis Techniques; Rubey, ARPA/DOD Conference, April 1976
4. Software Reliability Study; Thayer, Craig and Fray, TRW Defense and Space Systems Group, August 1976
5. Software Engineering Seminar; D. J. Reifer, Software Management Consultants, July 1979
6. DOD Weapons System Software Acquisition and Management Study; Asch, Kelliher and Locker, Mitre Corporation, June 1978
7. Software Requirements and Specifications; Ramamooth and So, August 1977
8. CORE - A Method for Controlled Requirement Specification; G. P. Mullery, Systems Designers Ltd. (to be presented at the IEEE Conference on Software Engineering, Sept. 1979)
9. PSL/PSA Application Guidebook; University of Michigan, Sept. 1977

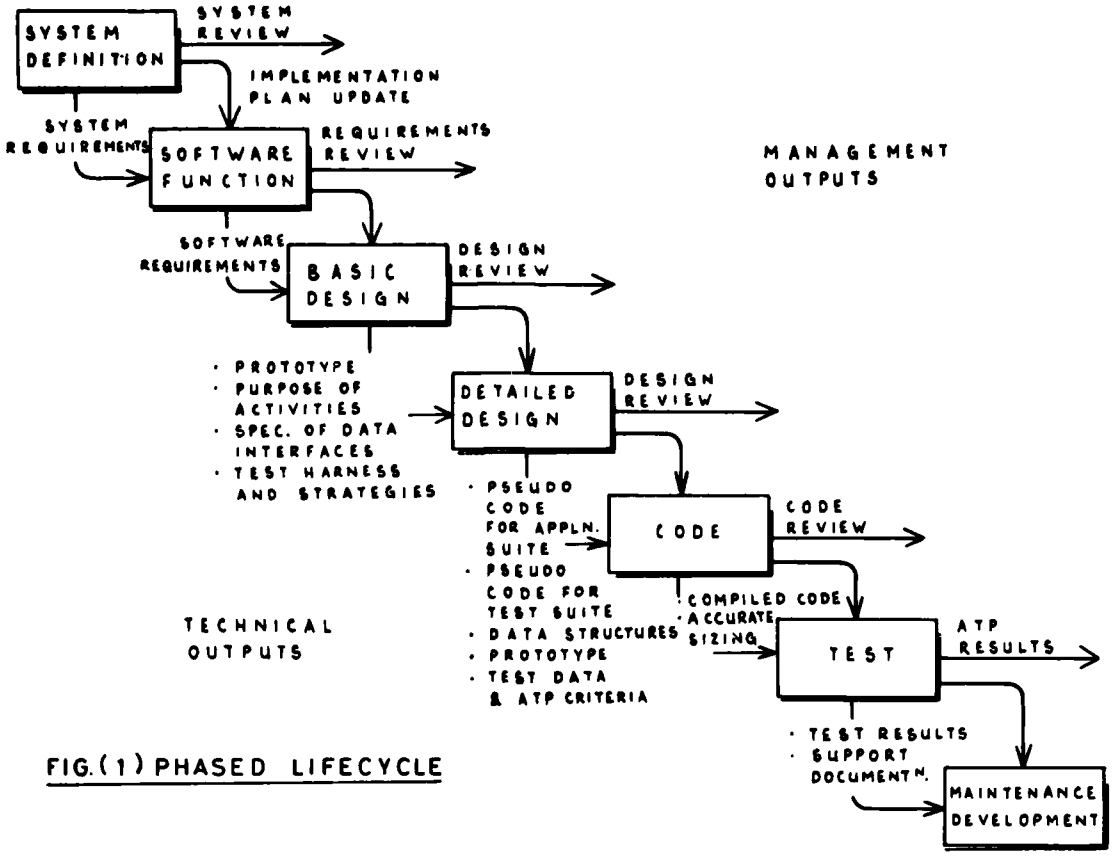


FIG. (1) PHASED LIFECYCLE

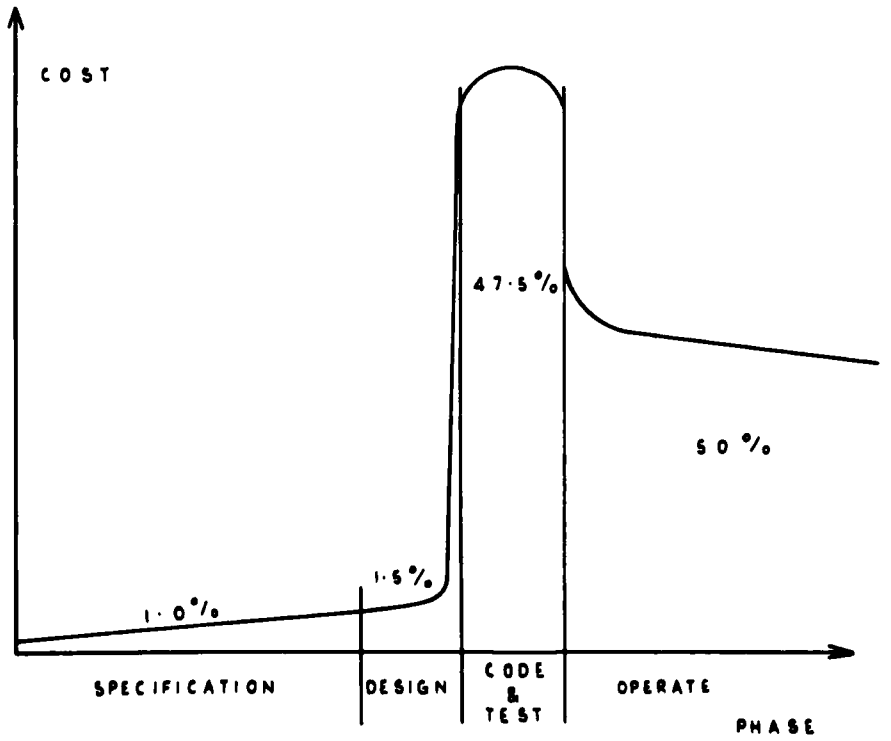


FIG. (2) DISTRIBUTION OF BUDGET OVER LIFE CYCLE

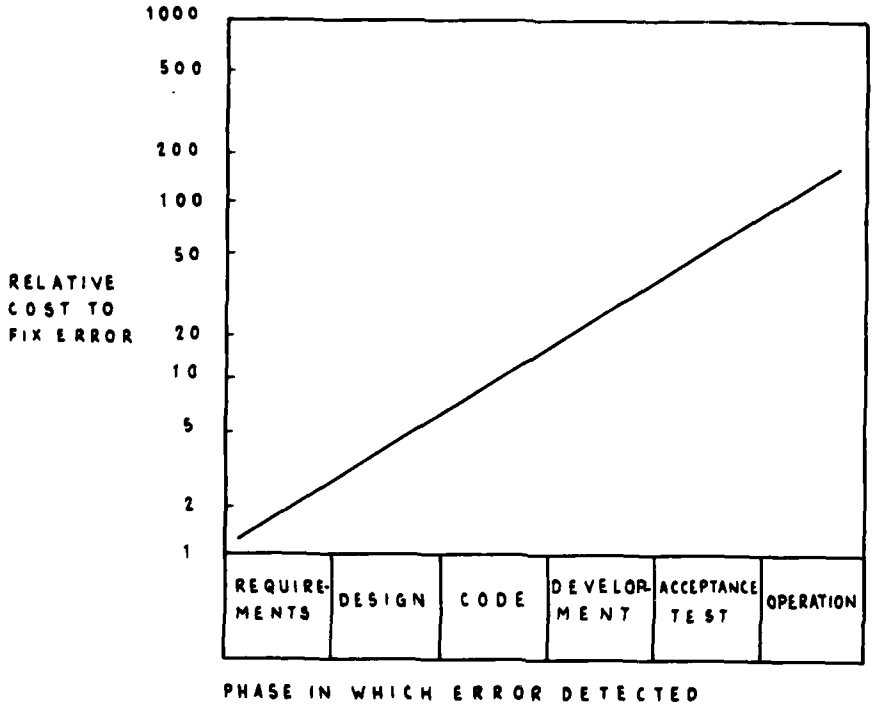


FIG.(3) RELATIVE COST TO FIX ERRORS OVER LIFECYCLE

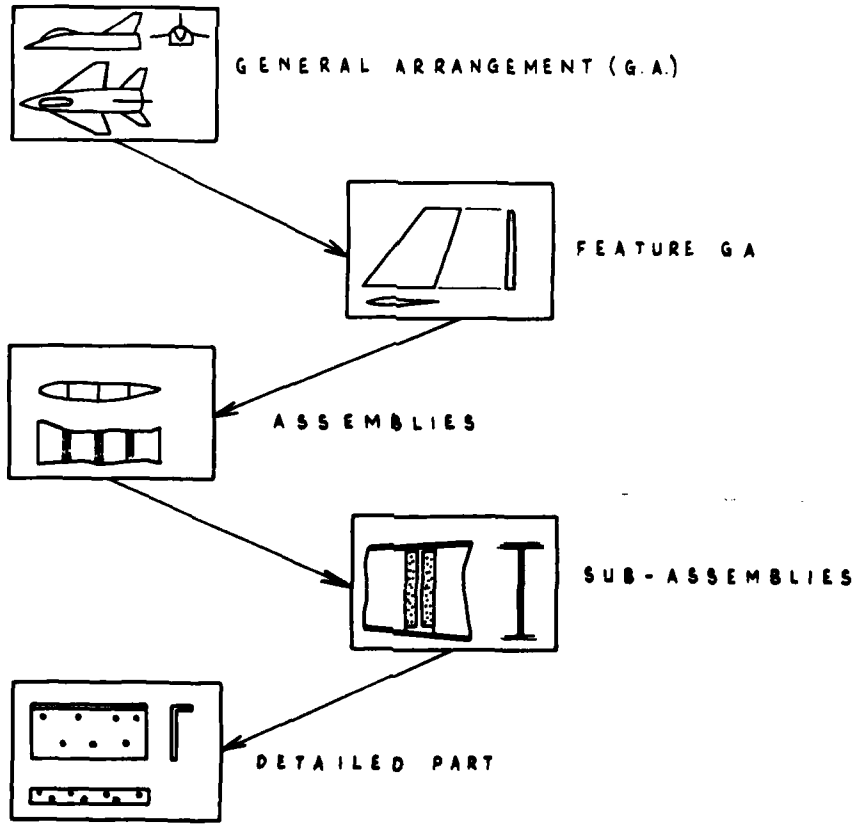


FIG.(4) NOTIONAL ENGINEERING DRAWING SCHEME

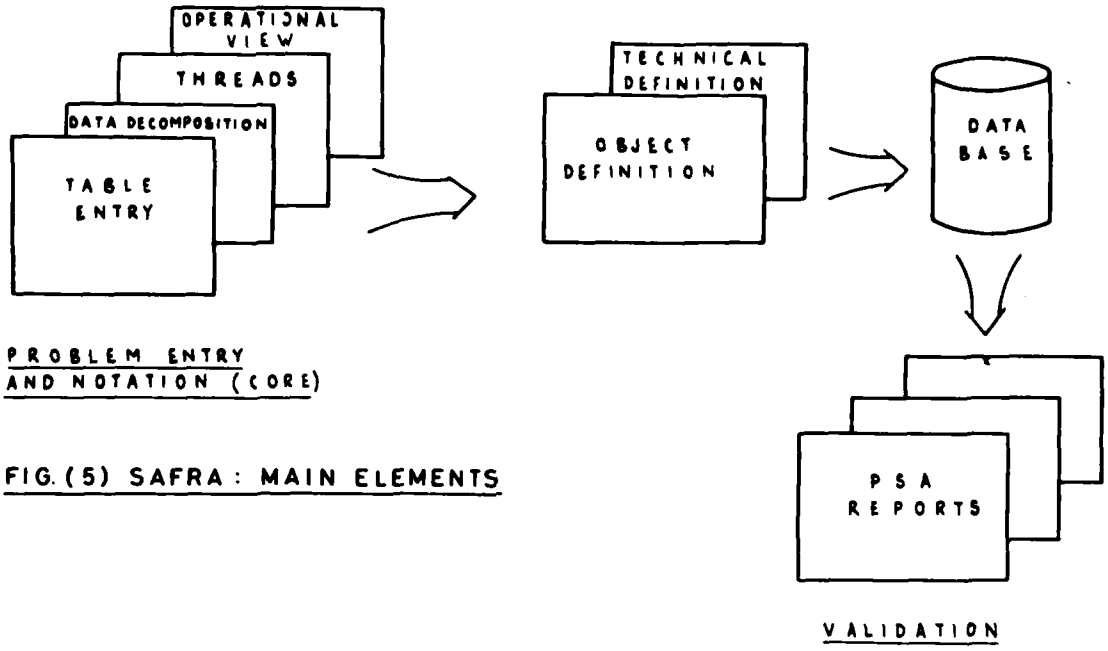


FIG.(5) SAFRA : MAIN ELEMENTS

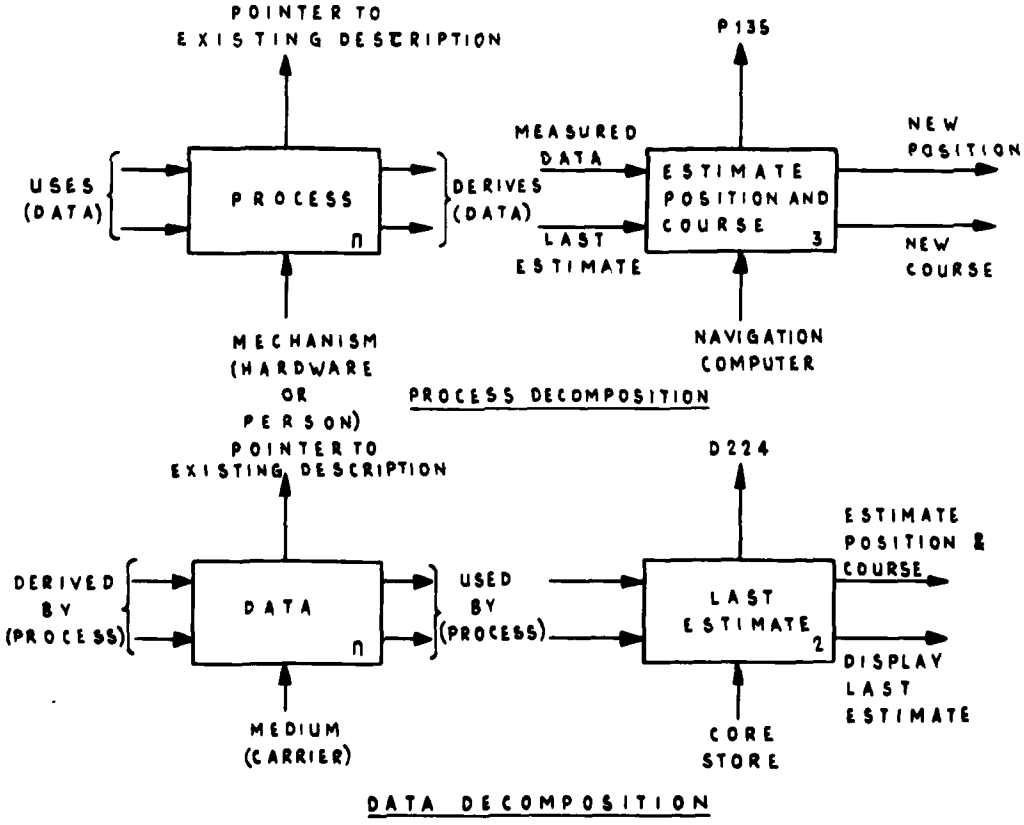
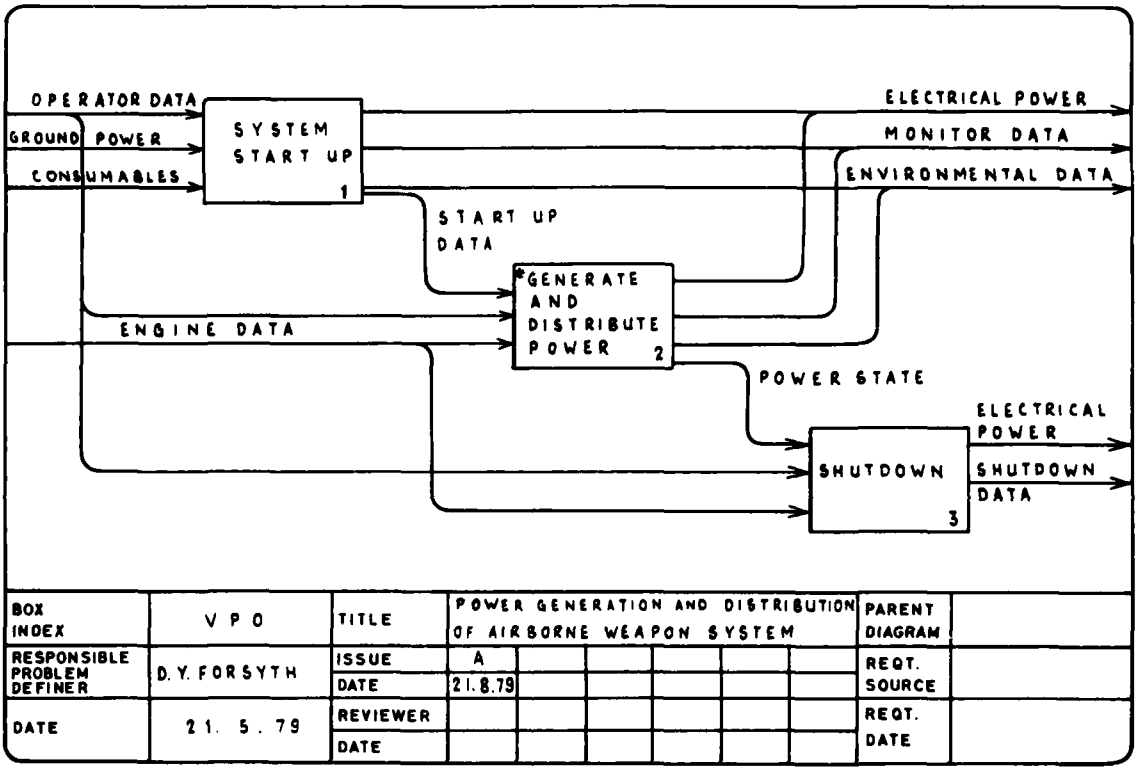


FIG.(6) PROCESS AND DATA REPRESENTATION



BOX INDEX	V P O	TITLE	POWER GENERATION AND DISTRIBUTION OF AIRBORNE WEAPON SYSTEM						PARENT DIAGRAM	
RESPONSIBLE PROBLEM DEFINER	D. Y. FORSYTH	ISSUE	A						REQT. SOURCE	
		DATE	21.8.79							
DATE	21. 5. 79	REVIEWER							REQT. DATE	
		DATE								

FIG. (7) DIAGRAM PROFORMA

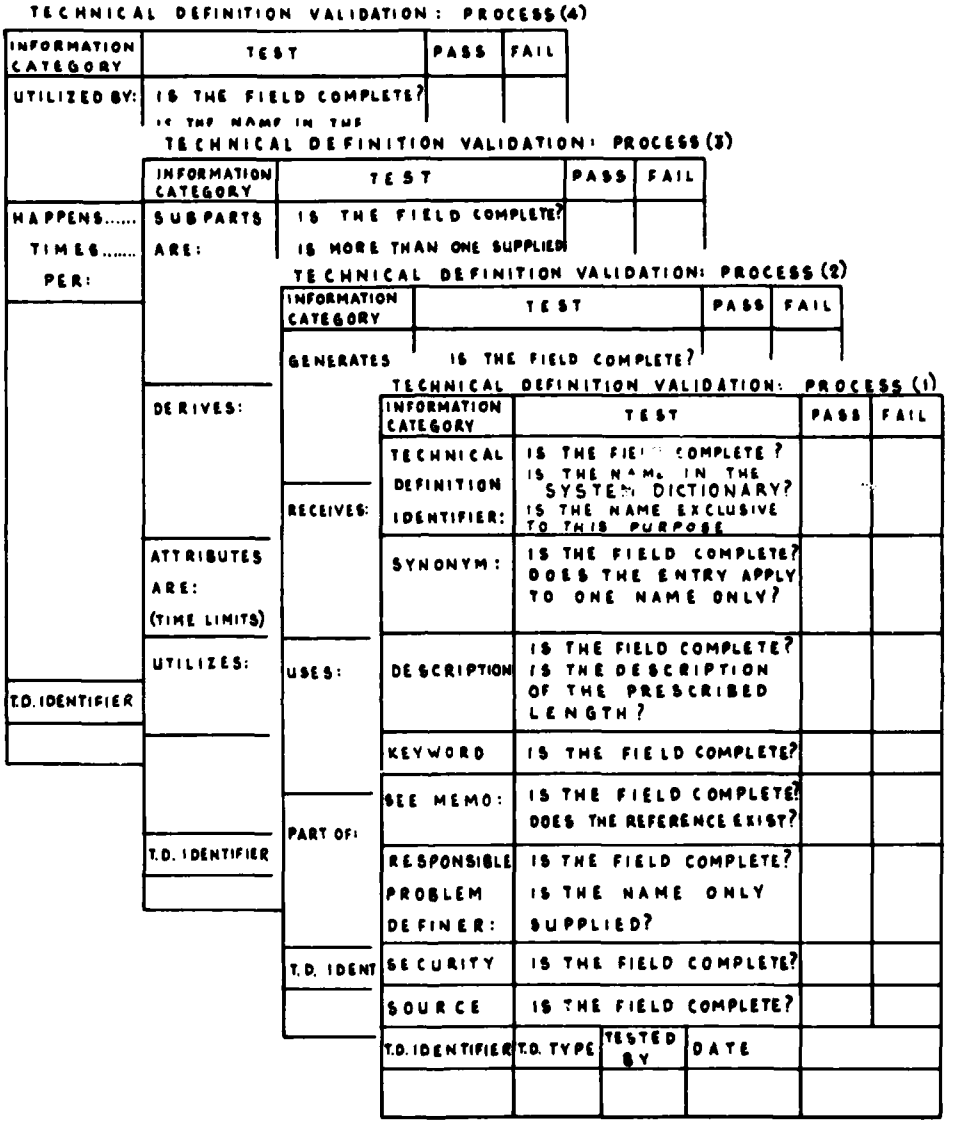


FIG. (8) VALIDATION CHECKLIST

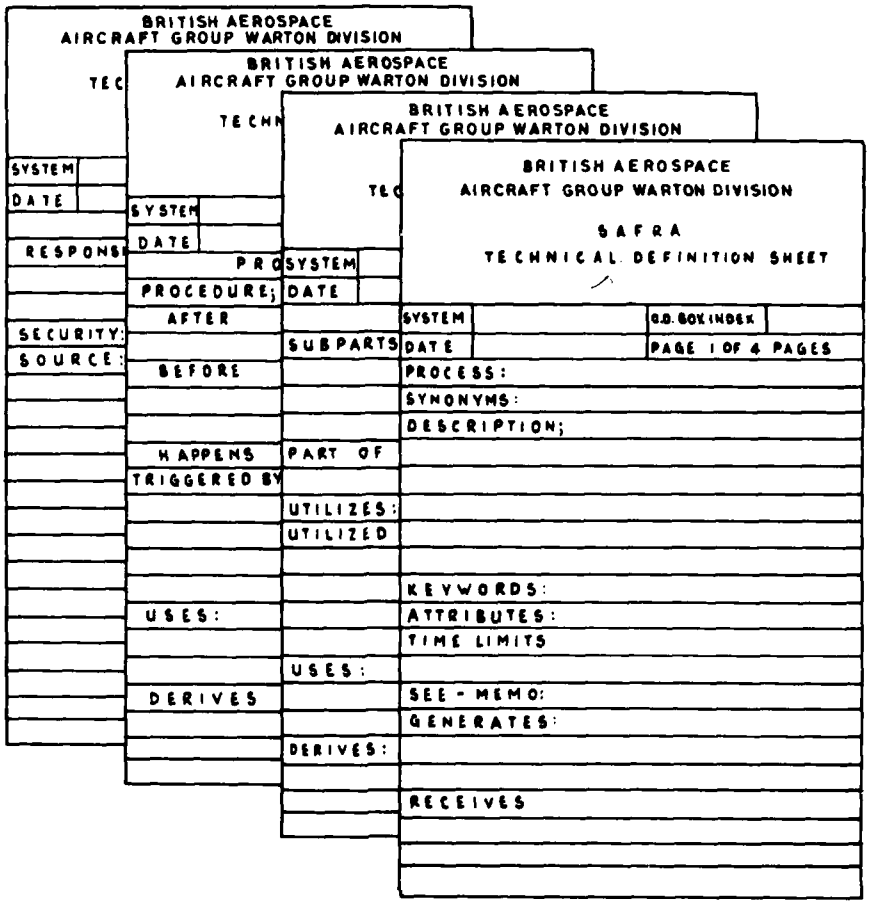


FIG. (9) PSL PROFORMA

```
PSA VERSION A4.2R1                                    79.135   19.59.44
                    BRITISH AEROSPACE WARTON
                        DICTIONARY REPORT

PARAMETERS :  DB = PSADB.DBF  FILE = PSATEMP.PSANAME  NOINDEX
NOPUNCHED-NAMES  NODESCRIPTION  SYNONYMS  NOKEYWORDS
NORESPONSIBLE-PD  NOATTRIBUTES  SPACING = 2  NONEW-PAGE PRINT

1.  A-16-BIT-DEDICATED-WORKSPACE      GROUP
                    SYNONYMS: A16WSPACE
2.  A-16-BIT-SCRATCHPAD-CONSTANT      GROUP
                    SYNONYMS: A16S-PC
3.  A-16-BIT-SCRATCHPAD-VARIABLE      GROUP
                    SYNONYMS: A16S-PV
4.  ADDITION-BOX-CLJ1                 PROCESS
                    SYNONYMS: CLJ1
5.  ADDITION-BOX-CLJ1-START-EVENT     EVENT
6.  ADDITION-BOX-CLK1                 PROCESS
7.  ADDITION-BOX-CLK2                 PROCESS
8.  ADDITION-BOX-CLL1                 PROCESS
9.  ADDITION-BOX-CLL2                 PROCESS
10. ADDITION-BOX-CLL3                 PROCESS
```

FIG. (10) DICTIONARY REPORT

PROCESS :	RESPONSIBLE-PROBLEM-DEFINER:
ATTRIBUTES ARE:	SECURITY IS :
DERIVES:	SEE-MEMO :
DESCRIPTION :	SOURCE IS :
GENERATES:	SUBPARTS ARE :
HAPPENS: TIMES-PER:	SYNONYMS ARE:
INCEPTION-CAUSES:	TERMINATION-CAUSES :
KEYWORDS ARE:	TRIGGERED BY:
MAINTAINS :	UPDATES : USING :
PART OF :	USES : TO DERIVE :
PROCEDURE:	UTILIZED BY :
RECEIVES:	UTILIZES:

FIG.(11) PROCESS SECTION: PERMISSIBLE STATEMENTS

```
PSA VERSION A4.2R1                          79.137.00.38.51    PAGE 1
                        BRITISH AEROSPACE WARTON

                    FORMATTED PROBLEM STATEMENT

PARAMETERS: DE = PSADE.DBF. FILE = ADV. PAB. JAG. DATA (NAMES1) NOINDEX
NOPUNCHED-NAMES PRINT EMPTY NOPUNCH SMARG = 5  NMARG = 20
AMARG = 10  EMARG = 25  RNMARG = 70  CMARG = 1  HMARG = 40 NODESIGNATE
ONE-PER-LINE  DEFINE COMMENT NONEW-PAGE NONEW LINE  NOALL-STATEMENTS
COMPLEMENTARY-STATEMENTS  LINE-NUMBERS PRINTEOF DLC-COMMENT

1   PROCESS                  LIMIT-L14;
2         /* DATE OF LAST CHANGE - 79.110.17.54.54 */
3         SYNONYMS ARE: L14:
4         DESCRIPTION;
5                          PITCH STICK COMMAND RATE LIMIT
6                          STORES LAST STICK COMMAND
7                          CHECKS PRESENT STICK COMMAND AND LIMITS RATE OF
8                          CHANGE TO 20 DEG/SEC
9         GROUP            ACCUML   16 BIT REGISTER
10                         CLQWS1   16 BIT DEDICATED WORKSPACE
11                         CLQC02   16 BIT SCRATCHPAD CONSTANT;
12    SEE-MEMO:            JAG-FBW-64;
13    PART OF:             CONTROL-LAW-MODULE-CLQ;
14    USES:                A-16-BIT-SCRATCHPAD-VARIABLE
15                         A-16-BIT-SCRATCHPAD-CONSTANT
16                         A-16-BIT-DEDICATED-WORKSPACE
17                         ACCUML,
18                         CLQWS1,
19                         CLQWS2,
20    DERIVES:             ACCUML,
21    PROCEDURE;
22                         AFTER   ADDITION-BOX-CLQ2
23                         BEFORE  FILTER-F12
24          EXECUTES
25                         WS1601 = ACCUML
26                         WS1602 = ABS((WS1601-CLQWS1)*CLQC02)
27                         IF (WS1602 > 20) THEN CLQOP1 = CLQWS1+(20/CLQC02)
28                                              ELSE CLQOP1 = WS1601
29                         CLQWS1 = CLQOP1
30          WHERE
31                         CLQC02 = 50    MODULE FREQUENCY
32                         CLQWS2         LAST CLQOP1;
33          HAPPENS:
34                  50 TIMES-PER SECOND;
35          TRIGGERED BY: LIMIT-L14-START-EVENT;
```

FIG.(12) FORMATTED PROBLEM STATEMENT FOR A PROCESS

```
P S A  VERSION A4. 2R1                                  79. 144 00. 59. 24  PAGE 5
                        BRITISH AEROSPACE WARTON
                          FORMATTED PROBLEM STATEMENT
 83    PROBLEM- DEFINER              D - TALBUTT;
 84          /* DATE OF LAST CHANGE   79. 110, 17. 54. 54 */
 85       RESPONSIBLE FOR:
 86                              CONTROL -LAW - MODULE - CLK.
 87                              CONTROL -LAW - MODULE - CLR.
 88                              CONTROL -LAW - MODULE - CLM.
 89                              CONTROL -LAW -MODULE - CLQ.
 90                              CONTROL -LAW -MODULE - CLN.
 91                              CONTROL -LAW -MODULE - CLL.
 92                              A - 16 - BIT - SCRATCHPAD - VARIABLE.
 93                              A - 16 - BIT - SCRATCHPAD - CONSTANT.
 94                              A - 16 - BIT - DEDICATED - WORKSPACE.
 95                              LINK - MODULE - L22.
 96                              LINK - MODULE - L23.
 97                              SCALING-FACTOR- G6.
 98                              ADDITION - BOX - CLJ1.
 99                              VARIABLE - GAIN - K11.
100                              FILTER - F13
101                              GAIN - SCHEDULE - GSM.
102                              GAIN - SCHEDULE- GUM.
103                              GAIN - SCHEDULE- GAM
104                              FILTER - MACRO - FTD.
105                              FILTER - F12
106                              FILTER - MACRO- FTB.
107                              GAIN - FACTOR - G5.
108                              ADDITION - BOX - CLQ1
109                              ADDITION - BOX - CLQ2
110                              LIMIT - L14;
111       /* 12 */
```

FIG.(13) PROBLEM DEFINER FILE

```
PSA  VERSION  A4. 2R1                                  79. 144  00. 59. 24  PAGE 6
                          FORMATTED PROBLEM STATEMENT
112 MEMO                          JTR - 072;
113       /*  DATE  OF LAST CHANGE -   78. 110. 17. 54. 54 */
114 APPLIES TO:              CONTROL-LAW- MODULE - CLK.
115                          CONTROL-LAW- MODULE - CLR.
116                          CONTROL-LAW- MODULE - CLM.
117                          CONTROL-LAW- MODULE - CLQ.
118                          CONTROL-LAW- MODULE - CLN.
119                          CONTROL-LAW- MODULE - CLP.
120                          CONTROL-LAW- MODULE - CLL.
121                          LINK- MODULE - L22.
122                          VARIABLE - GAIN - K11.
123                          FILTER - MACRO -FTD.
124                          FILTER- F12.
125                          FILTER- MACRO - FTB.
126                          GAIN-FACTOR - G5.
127                          ADDITION - BOX- CLQ1.
128                          ADDITION - BOX - CLQ2
129       /* 12 */
```

FIG.(14) MEMO FILE

TRENDS IN THE DEVELOPMENT OF SOFTWARE FOR GUIDANCE AND CONTROL

Dr. Peter F. Elzer
DORNIER SYSTEM GMBH
Postfach 1360
D-7990 Friedrichshafen
Fed. Rep. of Germany

ABSTRACT

The paper tries to classify the steps of the development process for computer programs in guidance and control systems. It is then tried to identify possible methologies and support tools for development on each level and to assess the state of the art. Several existing methods are described.

1. INTRODUCTION

As time goes on, one reaches a better and better understanding of the thought processes which are involved in the development of programs of some size for guidance and control systems. This may sound trivial, but it obviously was not during the past decade, where no visible progress followed the bold statements of computer scientists in the late sixties: " ... we have to bring programming from the state of an art to that of a science ... ".

Several approaches were tried, which tackled the problem from different sides and, indeed, solved parts of the whole problem.

Some shall be mentioned briefly:

- Program development systems
- Simulation and emulation
- High order languages
- Problem oriented packages and languages
- Portability techniques
- Interactive programming
- Structured programming
- Specification languages
- Programming standards
- etc.

The problem was that some of these approaches competed with each other in places where there was no reason to do so, others were too costly to implement, others were falsly related to each other, etc. All that resulted in a "confused" picture which seemed to reflect no progress at all.

But "suddenly", i.e. within the past three years or so, a certain hierarchy of "levels" seems to crystallize, according to which the development process may be structured. Furthermore, first experience is available as to which of the above mentioned approaches solves the problems of which application area. It may still take some more years unitl a general understanding is reached as to what these levels exactly are, and by which means or tools one should go from one level to another. But a certain guess can be made already now, though the author would like to emphasize that it is still his personal view. It will also become obvious that the properties of the different levels are not yet equally well understood, and that in some places there is an abundance of tools whereas there are next to none in others.

2. THE LEVELS OF SYSTEM DEVELOPMENT

The title of this chapter has been deliberately chosen not to be restricted to programs alone, because it is the authors' conviction, that up (or: down) to a certain level of detail, hardware and software have to be treated alike. The neglection of this aspect has led to quite a bit of difficulties in the past: e.g. an information gap - sometimes widening to an unwillingness even to communicate - betwen the "hardware-man" and the "software-man", or to premature design decisions after which each "party" was stuck with problems which were supposed to make life easier for the other. In this paper it shall therefore be tried to abstract from the differences between hardware and software as long as it appears feasible.

What levels can be identified under such premises:

1. Understanding the problem
2. Describing the problem
3. Sketching a solution
4. Refining the solution
5. Identifying the resources
6. Making the solution work
7. Maintaining the resulting system

As can be seen later, the "size" of these steps (or the "distance" between the levels) is still somewhat unbalanced, but, as said before, the classification is a first attempt. On the other hand it may turn out that this apparent unbalance is based on the fact that some levels are just better understood than other ones, and consequently there is more information on them.

But let us look at them in order and some detail.

2.1 UNDERSTANDING THE PROBLEM

On the first glance this has nothing to do with "programming", "computer-science", "system design" or the like.

It seems to be purely a matter of the application engineer, physicist, officer, or whosoever tries to solve a problem, maybe by means of a computer. But it is a deceiving first impression. There are examples that e.g. a sales-engineer from a systems house discovered that there was no computer necessary to automate a certain process once it had been properly analysed and understood, or that control engineers tend to specify much too high scan rates for inputs "just to make sure".

There are other cases like this and they seem to indicate that there is an educational problem: Applications people will have to develop a "feeling" for the necessary "computer power" and for the peculiarities of the process of digitalisation, its advantages and limitations. It is obviously not sufficient that somebody, who understands the application and somebody, who understands computers get together and try to discuss how to solve the problem. There are thought processes which have to go on within one single persons mind.

It cannot be excluded that in the future "cookbooks" or even automated information retrieval systems will facilitate this process, but on the other hand it seems that there is so much creativity involved that first stage of the development process that automation cannot help very much.

But in order not to become to philosophic we will leave this subject now.

2.2 DESCRIBING THE PROBLEM

This is still an area which is rather remote from computer technology proper. During the centuries different crafts have developed different methods for the description of a project, the most famous one being the plan for a building - which also describes the solution - or the numerous kinds of verbal descriptions which are issued as a basis for "requests for proposals" for aquiring something.

But here computer oriented research can help. Results of research on languages as well as on the "information content" of descriptions might lead to less error prone, less ambiguous or more understandable descriptions of a given problem. It still seems to be necessary that such description methods are developed by people with a good understanding of the problem and a fair amount of knowledge of the descriptional power of any kind of formalized notation.

2.3 SKETCHING A SOLUTION

This step is often overlooked or its importance forgotten. But in traditional technical disciplines like architecture or mechanical engineering it plays a very important role. The methods applied here rely heavily on the human capability of processing a huge amount of data very rapidly if they are presented in graphic form. Certain conventions, like the three projection planes, allow skilled people - after relatively little training - to produce an image of the solution within their minds, i.e. develop an understanding for the properties and the feasibility of the object to be constructed.

The developers of automated systems still have a long way to go in this direction. They also suffer from a number of disadvantages which the traditional disciplines do not have. In the first place they deal with abstract objects, like "information" or "energy" or "momentum". Secondly they have to deal with the dynamic behaviour of systems. The latter problem could be tackled by developing something like a "projection on time", a rudimentary form of which may be the "transition diagrams" which are used in control theory.

Here the specific capabilities of computers might be utilized. A developer might suggest a formula, describing a certain solution, and the computer might simulate it and inform the developer of possible responses in a graphic form. This sounds pretty abstract for the simple reason that each discipline may require a different methology for such sketching. Examples are the automobile industry where computers already help to design cars by presenting the shape of mechanical parts in three dimensions responding to the designer's inputs with respect to modifications, or the semiconductor-industry, where the intricate patterns of integrated circuits are laid out in display screens and can be easily revised by the designers.

But additionally it appears necessary to develop a methodology for "system design sketches" which works without any computer support at all. On one hand it might in some cases be much more cost-effective and on the other hand it would avoid the psychological stress which each interactive computer system imposes on its user, by forcing him subconsciously to make faster decisions than necessary and giving him fewer choices than possible. It may be compared to the production-line, which improved the productivity of the individual worker immensely on the first hand, but developed formerly unknown negative effects in the long run. And why should a young technology not learn from the mistakes of an older one and retain the advantages of "hand-held-tools" where they are appropriate.

2.4 REFINING THE SOLUTION

We have now gradually slid into an area, where automated tools can help a great deal, and, in fact, do and have done so for many years.

Here, given an adequate formalized method of description for the solution, enough computing power to handle the necessary data, and proper interactive devices, the computer can take all the combersome and boring detailed work off the human user's shoulders, like e.g. repeating calculations with slightly different parameters, solving differential equations with different sets of boundary conditions, checking wiring diagrams for completeness, making sure that all applicable standards are met.

All that does not sound very much like the problems one is "used to have" in "systems programming". But this is only true on the first glance. Imagine a "system" (or: "program", or: "something") with a database which contains information on every available module of a given production line, about the ways how these can be interconnected, voltage levels, fan out conditions, response times of available program-modules, etc. Imagine further a program which is able to simulate the behaviour of a hardware-software configuration specified this way or imagine a simulation package that is able to show you the behaviour of a compound of parallel processes with given time constraints under simulated input, either statistically or problem-structure oriented.

You need not even to imagine! Such things exist and have been successfully used - but mostly for other purposes. New CPU's have been designed this way, the behaviour of operating systems simulated, the electrical properties of IC's verified. It is thus just a question of time until such tools will be available for the still less costly guidance and control applications, or they will become necessary once these applications become more complex and costly!

Once we have reached this level we can have reasonable confidence in the feasibility and performance of a system to be developed. We can now proceed to exactly identify the resources we need and make the whole structure work.

2.5 IDENTIFYING THE RESOURCES

It is fully intentional that this step is described before the actual "problem solving". It has also been intentionally put after the other steps because it is - very often - done as one of the first steps and then restrains the designers freedom in an inadequate way or is done partially after the next step and then tends to foul up the structure and clarity of the data representation of the solution.

To the author this process is so closely interwoven with parts of the next step that both should rather be regarded as parallel activities. Nevertheless for sheer reasons of the way how people tend to think and to solve their problems, every designer will still first make a guess about the hardware he is likely to need and then start putting it together and programming it. Much simpler and tougher constraints are delivery times and production time lags for hardware.

But despite these "facts of life" one should not forget that there have to be feedback loops in the development process.

On the other hand also the "soft resources" should be planned and identified here. Data areas, buffers, structures, etc. should be outlined and their size estimated. Existing support routines, programs in other languages are resources as well and should be identified and made available.

Automated tools can already help a great deal in doing this job. If the simulation in the preceeding step is any good, it already contains lists and functional descriptions of all necessary hardware and software components. It should therefore be possible, if not easy, to extract this information and use it for further purposes.

2.6 MAKING THE SOLUTION WORK

What we have now is a kind of a model of the problem solution. The step which follows is the one which is most widely known and into which most of the research and development in the computer field has gone until now.

It can be decomposed into:

- development of necessary additional hardware
- detailed program design
- coding
- component test
- integration and system test

It should be noted that in the context of this paper this is the first time that a hardware oriented activity has been identified as a separate item. It should further be noted that this particular item comes right here and not much earlier - as it usually does, which in turn usually results in problems at integration time. But we will leave it at that and consider the development of hardware modules of limited complexity as a well understood problem, which we do not have to discuss here.

2.6.1 DETAILED PROGRAM DESIGN

This work item is far from being easy. The traditional means - flowcharts - has proven quite helpful and adequate for programs of moderate size, but turned out to be next to useless for describing the parallel structure of programs for embedded computer systems, and is of an all too seducive softness which tends to turn sizeable programs into beautiful patterns of incomprehensible complexity. Together with the nearly unlimited possibilities of good assemblers this has resulted in programs which sometimes are works of art, but ...

This "state of the art" has been of concern to computer scientiest as well as practitioners during the past decade. They have come up with different solutions. The practitioners introduced "programming standards" within their organizations, i.e. mostly "cookbooks" containing e.g. upper limits of the size of program moduls, forbidden instructions, regulations on the use of certain instructions, limitations on branches, etc.

Computer science, on the other hand, mainly developed the methods of "structured programming". The sometimes raging battles about which constructions were good and which ones should be outlawed ("GOTO considered harmful") seem to have settled down on a reasonable level of (dis-)agreement and now allow transfer of ideas and methods developed into the area of the practitioners. Partly this has already happend with the "Structograms", developed by Nassi and Shneiderman [NSH 73] with Jackson's method [JAC 76], or the IBM-originated HIPO-Method [IBM 75]. In the area of realtime-programming it is about to happen, e.g. with the diagrams developed by the author [ELZ 77/2] or with the pool-channel-interface-method used in MASCOT [JAH 76].

One characteristic of these newer methods is that they can be automated, if necessary. An example for a useful computerization of the Nassi-Shneiderman diagrams is the programming aid "COLUMBUS" [WIT 74], which shows the versatility of the method by applying the same structuring principles to three different programming languages: FORTRAN, COBOL and Assembler.

This remark leads us to another very well established program development aid - high order languages (HOl's). They have been one of the oldest and most proven means to facilitate the design of programs, to structure them, to speed up coding, to facilitate testing, etc., i.e. they considerably improved the state of the art in programming in several ways already several years ago. In the areas of "conventional" programming FORTRAN, COBOL and ALGOL were introduced nearly twenty years ago and have since then undoubtedly helped to solve problems which otherwise would have been drowned in detail and effort if assembly coding had been the only available method.

The situation has, however, been different in the areas of programming for guidance and control applications and general systems implementation.

The essence of this is that there are now highorder programming languages for nearly all areas of computing and that some of them will even be (internationally) standardized which will further help to reduce proliferation at least to a certain extent. A survey on that area is given in two papers by the author [ELZ 77/1, ELZ 78].

But a closer look at the success of the HOL's reveales another interesting phenomenon: The main reason for their success has been their ability to raise the level of abstraction on which a program designer can think and work, or, to put it an other way, they reduce the amount of detail by which he is bothered at any given time of the design process.

This principle has, of course, been formulated very early and a number of languages have been designed, the most famous being ALGOL 68, which contained elements for allowing the user to define his own level of abstraction on which he wanted to work. Under other aspects it has been addressed e.g. by PARNAS in his proposals concerning specification and structuring and by Wulf in the ALPHARD approach [WLS 76]. It has also been of major concern during the development of "Ada", an HOL which has been developed for the US-Dept. of Def. for future use in all types of embedded computer systems [ADA 79, ELZ 79/2]. The aim has been that, with one basic language, a user or a user community can define all the levels of abstraction necessary for their design problems, thus unifying the design process at least in steps 4, 5 and 6.

The principle of raising the "level of abstraction" has also had an impact on other areas of computer applications for guidance and control. A vast amount of research effort has gone into the investigation of operating system principles and elements. Here, the work seems to converge, at least for mono-processor operating systems. The principles developed allow to describe the structure of operating systems on a level which is just right for the "traditional" HOL's. This in turn may have a positive effect on the whole design process. It will be possible to utilize some of the interactions between conventional language elements and operating system elements in order to achieve better and more efficient programs. In another paper [ELZ 77/1] the author has investigated this effect in more detail and tried to identify some consequences.

2.6.2 CODING

With the elaborations on high order languages we have already touched the next step, coding. In case assembly languages are used, the design process is often even physically interrupted here and the specification handed over to another person, the coder. But this is not a necessary step and we may well regard the tool, by which a higher level language is transformed into machine-useable form, as an "automatic coder". It is usually called a compiler, but as technology advances, one recognizes that this tool is in fact a whole toolbox, consisting of, e.g. a front-end compiler, a code-generator, a link-loader, libraries, a control-interface to the user or the operating system, may be a special integrated assembler etc.

It should be noted that in a development process like the one outlined in this paper, there is no place any more for the "coder" in the traditional sense, who transforms - by hand - programs from a form readable by humans into a machine readable form. Automation has finally reached a very high state here. But, and this is another interesting observation, one begins already to question whether one should go any further at this particular point or even back up a little and allow for more human intervention. One example may serve as an illustration: During the high time of belief in compilers a language, PL/I, was developed, which could be translated in such a way that even programs which were not complete in some sense could be transformed into runnable code which in turn produced results - but not necessarily the ones the user expected.

One has backed up from this "fully automated philosophy" and now requires that the user completely specifies what he expects the program to do before it can be translated. This in turn leads to some more recent considerations according to which the user should be able to interfere with the compiler a little more - of course only at specified points - e.g. in order to extract information which he may need later. This, together, with many other requirements for a software environment for a modern HOL, has been investigated within the "Ada" - project, too [PEB 79, ELZ 79/1].

As we now obviously have reached the summit of possible automation of the system development process, it can only go downhill, which, indeed, it rapidly does.

2.6.3 COMPONENT TESTING

Testing is one of the areas in system development, where art and intuition, patience and craftsmanship still play an important role.

On one side this is just a consequence of the not-yet-orderly state of program development, but on the other hand it has intrinsic reasons. There is automated test equipment for hardware components, but there is little of that around for software. Maybe the reason is simply that there are usually many pieces of hardware produced to one design, whereas software is mostly just one-of-a-kind, but sometimes the reasons are just sloppiness, pressure of time, lack of know-how, etc.

But such tools are not sufficient. Testing has to be done already during coding and has to be built into the design. If automatic testing is not feasible or not worth the effort, feedback loops in the coding process, like incremental compilation, might help. Furthermore, for guidance and control applications it is of extreme importance that interactive testing aids are built in the production system, which allow the user, though unplanned, to interact with the program under test on language level. There are successful examples.

As a future overall effort it might be well worthwhile to structure the whole design process in a manner that proper feedback loops allow testing of the design decisions made on a higher level already on the next lower level.

2.6.4 INTEGRATION AND SYSTEM TEST

This is quite similar to the previous chapter, except that we are still a little further downhill and that considerations on symmetry of design and testing apply still more: Testing has to be built into the design.

But, as things have a tendency to go wrong, and "the best laid plans of men and mice ...", too, there have to be tools for reconfiguring and replugging, for bypassing faulty components and keeping record of effects already investigated. Not to forget retesting after the "program has not been changed at all ... nearly".

2.7 MAINTAINING THE RESULTING SYSTEM

This is a sad story or an ugly duckling, which hopefully will be come a swan. If one may believe hearsay, and there are few reasons not to do so, maintenance is a dull job, staffed with people who either are incompetent or who become so. There are horror-stories about programs getting out of control after some time of maintenance, etc. On the other hand there are estimates that maintenance consumes from fourty to ninety percent of the total system lifecycle costs, mainly because of the above reasons. What can be done?

Surprisingly, nobody seems to know an answer. During a study, "VEPAS" [VEP 77] carried out by "Project PDV" in the FRG it turned out that everybody saw that there was a problem, but nobody had an answer. During a workshop on software tools, held under the auspices of the US-DoD-HOL-project, the same result emerged.

One thing seemed to become clear: Maintenance has also to be built into the design. The design has to be transparent, the documentation complete, correct and easily available, there have to be training courses, the maintenance people have to be better motivated, etc.

As this field has obviously been of total noninterest to computer scientists, one has actually to look how the practitioners do it and maybe extract some principles from it.

To make things worse, maintenance of programs has one particularily nasty quality in comparison to maintenance of hardware: it changes the properties of the object being maintained. And this property may demand for completely new procedures to cope with it.

2.8 A "NON - ITEM"

Until now, the design process has been described as to how it might look like at some time in the future. The tone has been more positive where things are known to be under way, more negative, where the difficulties are greater, or no solution is in sight.

In general it appeared that one integrated design methodology for guidance and control systems might eventually be developed.

Bus this shall not mean that the author believes that there should be one great big, integrated, automated, interactive design support system to do all this. There should rather be a toolbox, or a shop full of assorted tools for different purposes and for cases of different severity. But there should be one consistent underlying philosophy or craftsmanship on how to apply them.

If this can happen, we will be quite a way ahead. But first let us look at where current tools can help us.

3. EXISTING TOOLS

3.1 INTRODUCTION

This introduction shall also serve as a kind of disclaimer: The following section shall - and can neither serve as a comprehensive overview on known design and development aids, nor shall it give a judgement of different methods. If the reader is interested in more details he shall be referred to [IEE 77], where a considerable number of modern design and development aids are presented and partially compared. It shall just give a short impression of the power and areas of applicability of some of the methods and tools mentioned. It will mainly be tried to identify which steps in the development process can be covered and supported by which method.

The following concepts will be regarded:

1. The design principles of PARNAS
2. SADT
3. HIPO
4. PSL/PSA
5. Extended Structograms according to ELZER
6. PEARL

The choice looks rather arbitrary, but this is only partially true. The methods have mainly been selected - and ordered - according to the increasing level of detail which they are able to handle. It will turn out that each method covers a certain range of design steps. It should, however, be emphasized that they are not necessarily recommended as the best ones of their kind, but are rather regarded as prototypes or examples of a certain class of approaches.

3.2 THE DESIGN PRINCIPLES OF PARNAS

They consist of a broad range of recommendations, rules and methods which are based on fundamental discoveries concerning system structure and behaviour, hierarchies, system families, etc. One particilar part has been succesfully used as a program specification method. But it seems to be independent of any specific application. Interestingly it could as well be applied to the specification of hardware, because its basic principle is the description of system modules in terms of "black boxes", which have certain "inputs", which in turn can assume a certain range of input values. The modules also have "outputs" and "output values", accordingly.

The function which is performed by the module can be described in natural language as well as in some kind of programming language. Mostly an agorithm is specified, which relates output values to input values. Error and exceptional conditions can be indicated as well. The method is completely abstract and therefore capable of describing any level of detail. It is also completely recursive and allows to describe the behaviour of a whole system in one consistent terminology.

It is designed around the principles of "abstraction" and "information hiding" and allows to describe software systems in a way how they should be structured. The "input" and output" philosophy aids in identifying resources.

A graphic representation and a computerized form of this method are not known to the author.

3.3 SADT

This is a proprietary method of SOFTECH, Inc., and potential users can buy instruction material and courses from that company. The name expands to "Structured Analysis and Design Technique" and is centered around the basic idea that there must be a graphic representation for the description of the behaviour of any system.

The method turns out to be a rigid formalization of the thought processes which go on during the analysis and design of system and as such is independent of the application to computerized or software systems. It goes together with a defined structure of the project development team applying the method. The basic descriptive elements are rectangular boxes, each representing one function or element of a system of any kind, and arrows, connecting these boxes and showing interdependencies. The meaning of the arrows is rigidly standardized as input, output, control and mechanism (ICOM). Detailed information is provided by means of natural language inside the boxed and along the arrows. The diagrams can be nested to any depth, i.e. each box can be expanded into another diagram, according to the principle of "structured decomposition".

A very interesting feature of the method is that it is fully capable of describing the "duality" of a system structure, i.e. either describing a system as a compound of activities which are interconnected by data or as a model of data which are transformed into each other by activities.

Usually two sets of diagrams are produced, each completely describing the system under one of these two aspects and thus helping in identifying resources, too. There are attempts to produce computer support for that method.

3.4 HIPO

This method ("Hierarchy plus Input-Process-Output") was developed by IBM. Interesting enough, the starting point was that a method for facilitating maintenance was sought.

It is less abstract than SADT. It also relies heavily on graphic representations and even a template is provided to draw the system description diagrams. The programs are subdivided into "processes", represented by a box, which contains the description of the actions to be performed by this process. A second box to the left of this "process" contains the "input (data)" and another one to the right the "output". Data flow is represented by arrows. This basic diagram is supplemented by formalized tables for "extended descriptions" and tree-like "tables of contents". Extra symbols, like "magnetic tapes" and "display screens" show that the method is more oriented towards requirement analysis. Printed forms ("worksheets") are also available for the developer. The whole method is subdivided into slightly different "packages", which cover:

- initial design
- detailed design
- maintenance.

Though the rather broad variety of different symbols and the whole layout indicates that the method was initially developed as a documentation aid on a rather pragmatic basis, it is capable of supporting structured programming and hierarchical decomposition. Computer programs for automated support of the application of the method are available from IBM.

3.5 PSL/PSA

In contrast to the techniques described until now, this method is heavily computerized.

It consists of a "Problem Statement Language" (PSL) and a "Problem Statement Analyzer" (PSA). It was developed within the ISDOS-Project at the University of Michigan and has since then been used several times, e.g. on large defense programs. Like HIPO, it is also more oriented towards a solution rather than towards a problem analysis, but it is formalized to a much higher degree. The PSL is based on a model of a generalized information system. This system is supposed to consist of "objects". Each of these "objects" has "properties", which in turn have "property values". The connections or interrelations between the objects are described by "relationships". A "system description", which is obtained by using the PSL, is then input for the PSA.

This is essentially a program system which accepts statements in the PSL, and is controlled by a special command language. It constructs and uses an "analyzer data base" and outputs reports and messages of various kinds, e.g. summary reports, reference reports, analysis reports, data base modification reports, process chain reports, data process interaction reports etc.

The software for this method is available on several large machines, e.g. IBM 370, Univac 1100, CDC 6000/7000, etc.

3.6 STRUCTOGRAMS

This method was originally developed only as an alternative to traditional flowcharts, which proved inadequate for realtime programming and of too low level for use with HOL's [ELZ 77/2].

The original diagrams by Nassi/Shneiderman were developed for conventional programming only and consist of four elements:

1. code sequence
2. if-then-else (alternative)
3. case-statement
4. loop

The extensions for realtime purposes are:

5. parallel clause
6. synchronization block
7. integrated signal
8. exception handler
9. protoprocess

During development it turned out, however, that the underlying concepts had to be very carefully adapted to each other to ensure orthogonality (which is important for simple use) and coherence with the principles of structured programming. This led to the introduction of the principle of "virtual resources" which are basically access rights to actual resources and allow a clean separation of the description of the static and the dynamic parts of programs. It was also possible to identify the role of interrupts as "not-reusable resources "and to establish rules for the treatment of exceptions in process hierarchies.

The method turned out to be a promising means for sketching parallel program systems, but it also has the potential for computerization. Its constructs can be directly translated into HOL's and input for simulation packages can also be automatically derived.

3.7 PEARL

This is a high order language for process and experiment automation realtime applications [PER 78].

In this capability it was designed to facilitate parts of the detailed design and of the coding phase. But due to a unique feature, the "system description part", which allows to describe the hardware configuration of an automated system as a separate part of the program, it is also useful as a means to support system design in general and especially the identification of resources.

It facilitates test and maintenance insofar, as PEARL programs are much more readable than any Assembler. In fact, tests have shown that the main savings due to the use of PEARL can be achieved in the test phase.

4. SUMMARY

This paper is neither meant as a tutorial on program development methods nor as an outline of a plan for the development of an "all singing and dancing" program system (or even: method) for system development for guidance and control. It shall just support the author's view that practical progress in this area can be made in some reasonably near future.

The potential is there, prototype systems have proven successful, and it is now necessary to have a closer look at these to extract the elements which are fit for practical use and to combine them in a reasonably modular fashion.

5. REFERENCES

1. ADA 79
   Preliminary Ada Reference Manual and Rationale for the Design of the "ADA" Programming Language;
   SIGPLAN Notices, Vol. 14, No. 6. June 1979, (Part A and B)

2. ELZ 77/1
   Elzer, Roessler; Real-time Languages and Operating Systems;
   IFAC/IFIP Symposium on Digital Computer Applications to Process Control; Van Nauta Lemke Ed. The Hague, 1977, IFAC and North-Holland Publishing Comp., 1977

3. ELZ 77/2
   P. Elzer; Ein Mechanismus zur Erstellung strukturierter Prozeßautomatisierungs-programme;
   Informatik Fachberichte No. 7, GMR-GI-GfK Fachtagung Prozessrechner, Springer Verlag, pp. 137-148

4. ELZ 78
   Elzer; Efforts to standardize a high order language for realtime applications;
   IFAC-Congress, Helsinki, 1978

5. ELZ 79/1
   P. Elzer; Some observations concerning existing Software environments;
   published by US-DARPA, May 1979

6. ELZ 79/2
   P. Elzer; The evalution of the requirements for the software environment for "ADA";
   Proceedings of the first European Symposium on real-time data handling and process control, Berlin (West), 1979

7. IBM 75
   HIPO - A design Aid and documentation technique;
   IBM GC20 - 1851 - 1, 1975

8. IEE 77
   IEEE transactions on Software Engineering;
   January 1977, Vol. SE-3, No. 1

9. JAH 76
   Jackson, Harte; The achievement of well structured software in realtime application;
   Proceedings of "IFAC-IFIP" international workshop on realtime programming, IRIA, Roquencourt, 1976, S. 145-15

10. JAC 76
    M.A. Jackson; Constructive Methods of Program Design;
    Proceedings of the first Conference of the European Cooperation in Informations, 1976;
    Lecture notes in computer science, No. 44, Springer Verlag, 1976, pp. 236-262

11. NSH 73
    Nassi, Shneiderman; Flowchart Techniques for Structured Programming;
    SIGPLAN notices, Vol. 8, No. 8, Aug. 1973, pp. 12-26

12. PAR 72
    D.L. Parnas; A technique for Software module specification with examples;
    CACM, Vol. 15, No. 5, May 1972, pp. 330-336

13. PER 78
    Basic PEARL;
    DIN, Draft Standard 66253 (Part 1)
    June 1978, Beuth Verlag, Berlin, Köln

14. PEB 79
    US-Department of Defense, Requirements for the Programming Environment for the common HOL;
    "PEBBLEMAN revised", published by DARPA, Jan. 1979

15. VEP 77
    "VEPAS"; Verfahren zur Erstellung von Prozeßautomatisierungssoftware;
    PDV-Entwicklungsnotiz, 1977

16. WIT 74
    J. Witte; The structured programming engine;
    ACM German Chapter, Lectures III, 1974, pp. 47-74

17. WLS 76
    Wulf, London, Shaw; An Introduction to the Construction and Verification of ALPHARD programs;
    IEEE Transactions on Software Engineering, SE-2, 4, Dec. 197[illegible], p. 253-265

A MODERN FACILITY FOR SOFTWARE PRODUCTION AND MAINTENANCE

by
H. G. Stuebing
Superintendent, Advanced Software Technology Division
Software and Computer Directorate
U.S. Naval Air Development Center
Warminster, Pennsylvania 18974
United States of America

SUMMARY

A facility has been designed, developed, and used for the life-cycle support of weapon system software. This facility consists of a software system which runs on a commercial multicomputer configuration. The approach features increased management visibility of the software development process, increased programmer productivity through automation, reducing the cost-of-change during maintenance, and the use of automated regression testing to improve software quality.

This paper describes the underlying issues which guided the development, provides an overview of the operation, and discusses the experience gained in implementing and using the facility.

1. BACKGROUND

SOFTWARE ENGINEERING

The main concern of this paper is the manufacturing of software or, more specifically, the economical manufacturing of a software product which is reliable, efficient, and functionally satisfactory to the end user. The software product is a large-scale software system which executes on military computers and provides a specific operational capability.

In order to orient the reader for the main section of this paper, a short discussion on terminology is necessary. In this paper the term "software" refers to a set of computer programs written by professionals which are intended to be used by others. The term "computer program" refers to a series of statements prepared in order to achieve a specific result. In a broader sense the term "software system" refers to a collection of computer programs, procedures, and methods which interact in an organized way to accomplish a set of specific functions.

The phrase "software engineering" is used in this paper and a short discussion is needed to put this phrase in proper context. When the phrase "software engineering" first appeared, it was used in a provocative sense and was intended to draw attention to the fact that software manufacturing was not based on the theoretical foundations and practical disciplines that are traditional in the established engineering fields. "Software engineering" was further intended to emphasize a concern for a product which works; where working means meeting commitments of function, cost, and schedule. The phrase "software engineering" was intended to contrast with the phrase "computer science"; the latter aims at defining general principles underlying the design and application of computer systems (including both hardware and software). A working definition of "software engineering" is that it is concerned with developing software systems that satisfy the requirements of the user over the life of the system.

SOFTWARE PRODUCTS

The software for a weapon system can be categorized as follows:

Operational Software
System Test Software
Support Software
Software Documentation

The Operational Software is a software system which executes on a military computer and performs many critical functions for the total weapon system; typical functions for an airborne system are tactics, navigation, sensor processing (sonar, radar, etc.), target tracking, weapon control, man-machine interaction and many more. System Test Software is an independent software system which executes on a computer prior to mission use; this software determines the operational readiness of the computer and its peripheral subsystems. The size of the Operational Software and System Test Software in terms of computer storage are roughly equal.

The Support Software consists of a set of computer programs which are required to develop the Operational Software and System Test Software; typically, it consists of high-level language compilers, assemblers, software emulators or simulators, system generators, debug-aids, software engineering tools, and an operating system. Originally, all Support Software was developed and executed on military computers in a laboratory facility. Support Software is not delivered to the operational forces but is delivered by the development agency to the designated maintenance agency. Depending on the particular project there are other categories of software such as weapon system trainer software and software for Automatic Test Equipment (ATE).

Software Documentation is produced for Operational Software, System Test Software, and Support Software. The documentation is produced in accordance with formal standards and specifications and may exceed 15,000 pages for a large-scale software development.

SOFTWARE LIFE-CYCLE

The use of the term "life-cycle" implies a concern for the product from the time of conception to disposal. Many weapon systems have a system life of 15 to 20 years. The software life-cycle covers the same period as the system life-cycle.

From a system's viewpoint the software is one of many components or subsystems. The software life-cycle can be broken down into distinct sequential phases of activity as shown in figure 1.

The purpose is to define those activities which have measurable inputs and outputs, thus allowing management review and control. The activities overlap in time to show that there is an interaction between the activities. There are iterations both within activities and across several activities depending on the nature of the project.

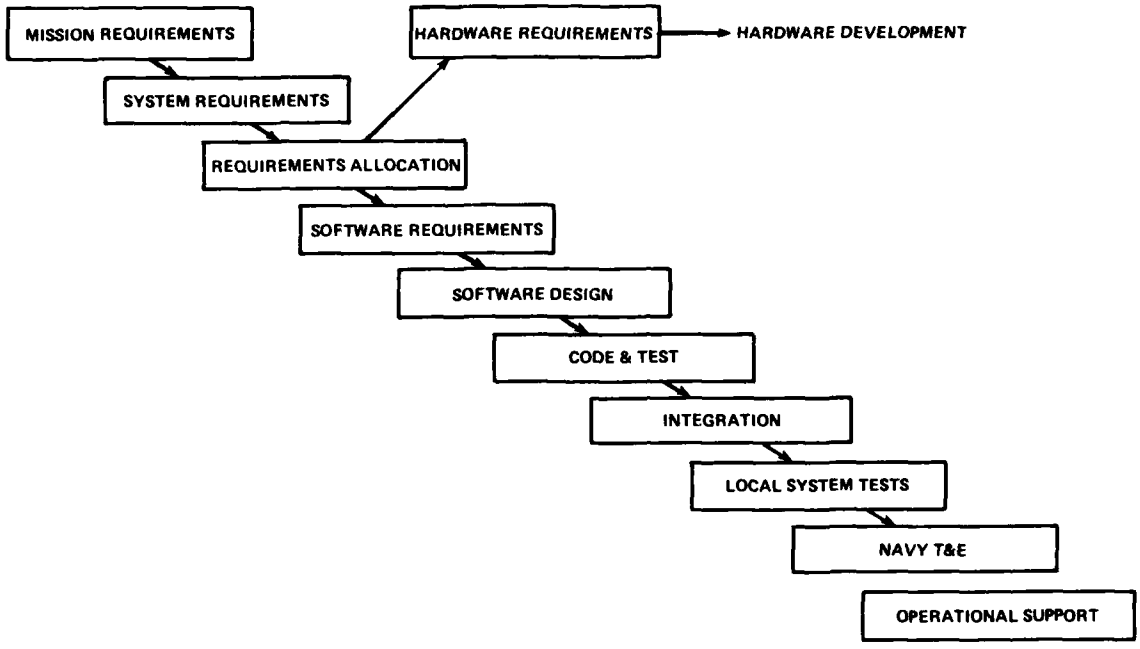


FIGURE 1 – Generic System Development Process

2. SOFTWARE PROBLEMS

The complexity and cost of weapon systems have increased dramatically over a period of several years. Concurrently, the use of digital computers also increased because the programmability offered a means of extending the useful system-life by permitting adaptation to new threats or tactical situations. Building large, complex weapon systems at affordable costs has been a major problem; historically, the development of both hardware and software systems has been poor from a cost performance viewpoint. There is an optimism concerning hardware system costs because technological breakthroughs are continually reducing the component costs. However, the recurring software problems of late deliveries, poor quality, and especially increasing life-cycle costs have created a somewhat pessimistic attitude regarding software.

These software problems have been discussed at length in reports, professional journals, and national trade publications; numerous committees, study groups, and symposiums have also addressed the problems. References 1 through 8 document the software problems.

ANALYSIS

Studies of weapon system life-cycle costs have shown that about 25% of the total cost is spent during development, the time from system concept to operational deployment. Thus, 75% of the total cost is spent on maintenance, a term which covers activities from minor error corrections to large functional enhancements. It is clear that to substantially reduce life-cycle costs the software must be developed in a manner which makes it easy to change during maintenance.

The weapon system acquisition environment is an important factor since the life-cycle support is a shared responsibility between Department of Defense (DOD) and industry. The long-term expertise in military problems rests in the government laboratories; therefore, the solution to the software life-cycle cost problem requires a management strategy which blends the vast talents and resources of industry with the long-term expertise of the laboratories.

There are some important distinctions between the use of computers in weapon systems and the use in most commercial and industrial applications. In the latter case a programmer that discovers a problem with his software has the capability to quickly diagnose the problem, correct the program, and conduct new tests; this capability exists because the software is developed, tested, and executed on the same computer. This capability does not exist for computers in weapn systems because of the constraints in the operating environment. The military computer on which the program is executed does not have an adequate programming environment for development and testing. Another distinction is that the software for a weapon system is distributed to a large number of operational units and usually there are small but critical differences between each version; these differences are due to different armament, sensors, or tactics. Further, weapon system software is typically a real-time application which means special consideration must be given to computer storage space and execution time. These distinctions mean that weapon system software must be carefully designed, developed, thoroughly tested, and distributed with variations to deployed units. Additionally, an environment must be prepared in which the software can be efficiently maintained.

The use of many different types of computers, both within and across weapon systems, has been a major source of software problems. Perhaps equally significant was the policy of developing and maintaining the software on military computers. This created a situation where new military computers were constantly being introduced and before they could be used a completely new set of support software would have to be developed. In other words, the tools were being developed at the same time as the final software product.

Also significant was the lack of recognition that software was a labor-intensive field; it was not a well developed engineering discipline. Successful software developments at reasonable costs were rare; there was no methodology which could repeatedly produce high-quality software on time and at affordable costs.

The software problems did not have a single source; there was no single change which would solve the problems. In fact, every activity of figure 1 needed improvement. The software problems were caused by technical inadequacies in the software field itself, as well as management's inability to adopt the correct policies and enforce their use.

3. FACILITIES FOR LIFE-CYCLE SUPPORT

To produce better weapon system software at significantly reduced life-cycle costs requires an overall strategy which blends the talents of government laboratories and industry. The first step is to partition the initial development effort into two steps – software production and integration. Next, facilities for each function are designed and constructed to support the weapon system over the total life-cycle. The engineering activities during the maintenance phase require the same facility features as the development phase, making it possible to do a design for the total life-cycle.

Industry, in a competitive acquisition environment, uses these facilities to produce and maintain the software with long-term expertise in the weapon system being provided by the government laboratory. Since the capital investment is high it is intended to share the facilities across projects although this is generally only possible with the software production facilities.

An important part of the strategy is to adopt an engineering methodology which is supported by the facilities. The activities shown in figure 1 can serve as an example; the software development, and later the maintenance, is broken down into work activities each with a measurable input and output. Many of the activities use the facilities and rely on the particular features which are implemented. In contrast to the development of a hardware system the end result of a software development is an intangible product which can only be observed indirectly through the associated documentation. Thus, all inputs and outputs of the activities are documents, in great quantity and of various forms.

SOFTWARE PRODUCTION AND MAINTENANCE

The software end products are dependent on the support software, e.g., compilers, assemblers, etc.; thus, the software production function offers a natural leverage point to improve all software problem areas. The strategy used for this facility was:

- use modern commercial computers for the host facility;
- support several large projects concurrently;
- increase management visibility and provide a means for measurement and control;
- increase programmer productivity by enhancing the programming environment and automating many of the functions;
- use software emulation of the target computer to develop the "software first";
- use the new concepts of structure and modularity to reduce the cost-of-change during maintenance; and
- use a flexible superstructure in order to easily accommodate new software engineering tools and techniques.

INTEGRATION FACILITIES

Integration Facilities exist for each project and consist of laboratory hot mockups of the actual military computers with realistic simulation of external signals.

The Integration Facilities serve as the hardware configuration baseline and are used for hardware/software integration, evaluation of man/machine functions, and for test and evaluation of Engineering Change Proposals (ECP's). The simulation of realistic external signals allows the total system to be tested in a laboratory environment with proper instrumentation and easy access to the equipment. This approach minimizes costly field testing, a very difficult environment to test hardware and software. Initial tests are performed locally prior to formal test and evaluation.

4. FACILITY FOR AUTOMATED SOFTWARE PRODUCTION (FASP)

This section contains a generic description of FASP, a software system implemented according to the previously stated strategies. FASP is implemented at the U.S. Naval Air Development Center (NADC), Warminster, PA, a Navy laboratory with life-cycle responsibility for several major weapon systems.

The initial studies began in July 1972 and the first operational version was put in use in July 1975; FASP has been used continuously since that date. The first version was named the Software Engineering Facility (SEF) but as the software system evolved the name was changed to FASP. References 9 and 10 contain additional information.

During the early 1970's, NADC was faced with a rapidly expanding software workload; the work was for several major projects and included both development and maintenance responsibilities. At that time it was normal for each project to have a separate software development facility with the support software executing on a military computer.

At the same time the software problems of cost, delivery schedule, and quality were gradually being recognized at the highest management levels. Thus, in a project production environment, where the constraints of time and money were rigid, the problem became one of introducing radically new methods while minimizing risk to the project.

The overall approach was to select a single project and to phase the development of the new support system to meet the critical schedule dates. Although the analysis of software problems indicated that the entire software development process (figure 1) needed new methods, only the code and test phase was initially addressed. The code and test phase was selected for three reasons: first, there were no proven tools for the requirements and design phases and the development of such tools was judged too high a risk; second, the code-and-test activities were well understood and were, therefore, easier to adapt to automated methods; third, once a project used the new facility for code-and-test the software was "in-place" for maintenance, clearly an advantage to developing the software on a separate system and later performing a conversion.

ARCHITECTURE

From a functional, or user's, point of view, FASP is built around a project data base, a related collection of computer accessible libraries. The data base contains not only the actual project software but a wealth of supporting data and management information. The user interfaces with FASP through a command language, invoking various types of programs which interact with the data base. These

programs, sometimes called "tools", are editors, compilers, system generators, simulators, test analyzers, etc. FASP is a highly-integrated system since all the internal operations between the tools and the data base are transparent to user; the user sees only the command language.

FASP is both an advanced programming system and a management information system. As a programming system FASP provides a set of uniform procedures and certified tools for developing and maintaining weapon system software. As a management information system, FASP provides on-line access to a variety of production data which has been automatically stored in the data base.

The FASP software architecture is shown in figures 2 and 3, the second being an expansion of the first. The stacked boxes on the left of each figure represent the computer mainframe; the bottom box represents the basic hardware, the next highest the standard operating system. The FASP software system executes as an application program under the control of the operating system. The normal environment is multiprogramming so at any given time a very large number of FASP jobs are active in the system.

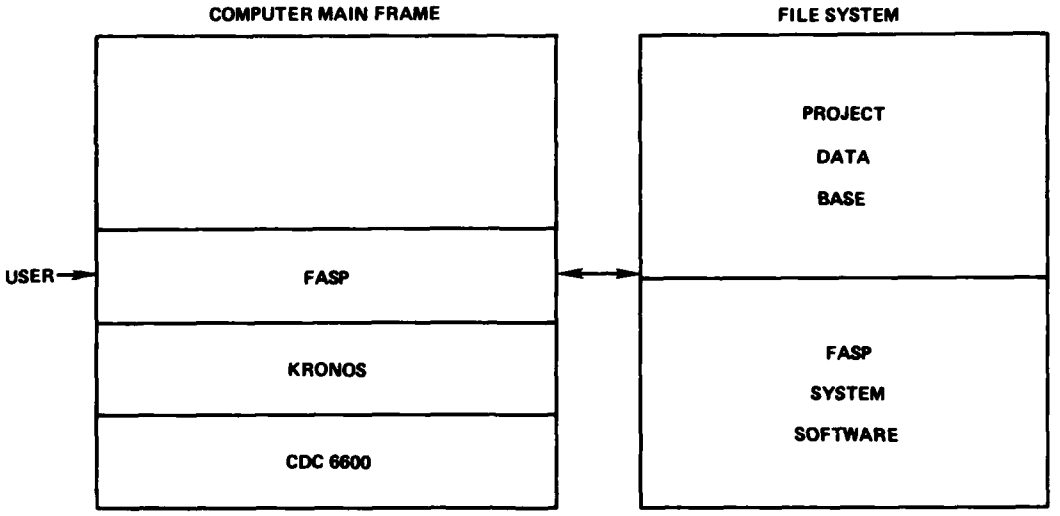


FIGURE 2 – FASP CCS (Central Computer System)

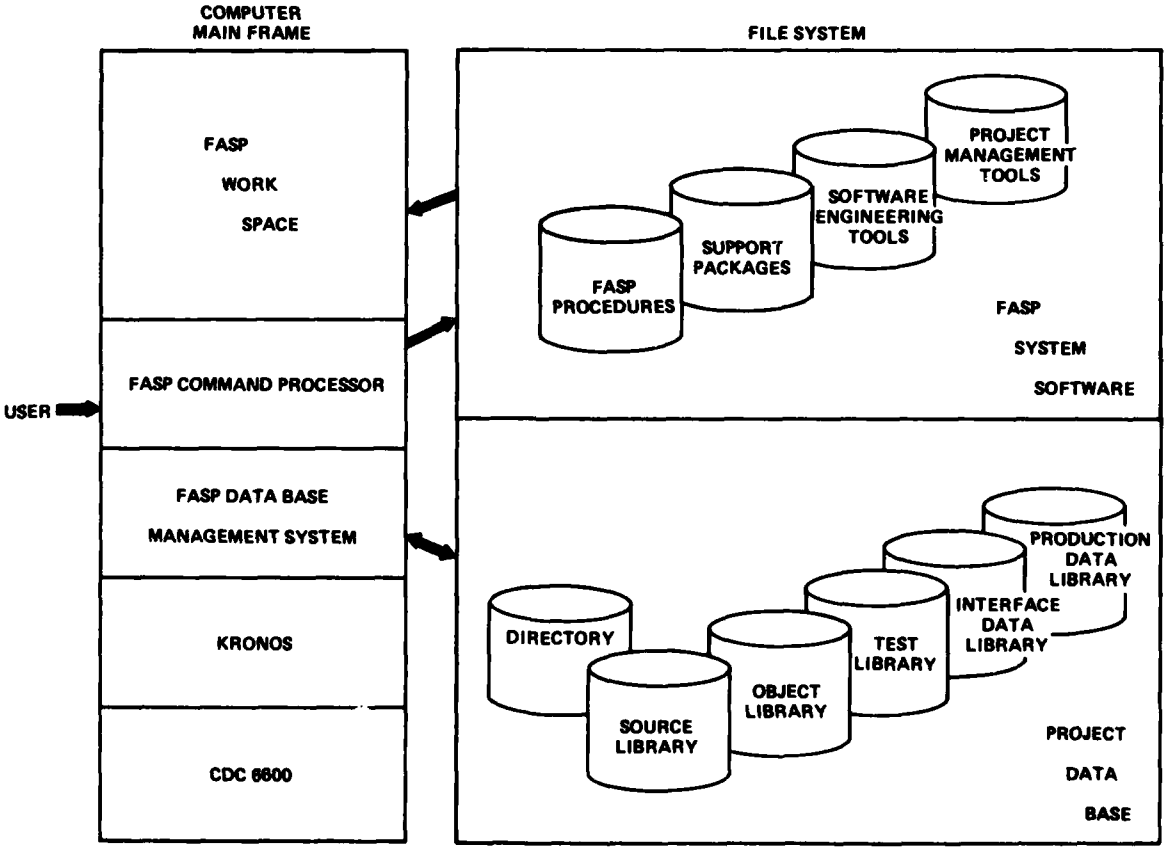


FIGURE 3 – Single User/Single Project FASP

FASP COMPONENTS

Project Data Bases

Under FASP, the developing weapon system software resides in the project data base. Normally, rather than one large data base the project software is distributed across several data bases in units of manageable size; the smaller units are then combined to obtain the total software. The data bases are not only repositories for project software but also are a source of technical and management information that reveals the genesis and current status of software production down to the module level.

Data bases are the basic means for independent development of separate pieces of project software. At the same time, the continuity of format and content over all data bases ensures the smooth integration of a complete system or subsystem.

Each data base is organized into a number of libraries; all are keyed to a source-code library that contains the most recent version of the project software. A master directory is automatically maintained for locating information in these libraries. As shown in figure 3, each project data base contains the following libraries:

(1) Source Library

The source library is a file containing the source code for each software module; source code refers to both programming language source statements and comment-cards which are interspersed throughout the program. In the FASP, certain standards and conventions are established for program comments so that FASP processors can extract certain data for reports.

Medium to large-scale computers of the third generation have extensive file handling capabilities where the data is normally stored on disk files and backed-up on magnetic tape. This capability allows the elimination of card decks as the principal storage medium and results in greater programmer and operational efficiency. Programmer efficiency is increased because of greater capabilities, such as common-decks (lines of source code which are shared across many modules); simple commands which allow additions, deletions, and editing of source code; and, audit-trail information which automatically records dates and times of changes to the source library. Operational efficiency is increased because large card-decks are not continually re-read (processing is done directly from disk files), disk files actually store card-images in compressed formats which reduces machine time and disk space, and back-up procedures minimize recovery in case of computer malfunctions or damage to source cards.

The source library is organized as software modules where a module is understood to be a program or part of a program which performs at least one complete function and which can be compiled or assembled as a unit.

When common-decks, which are blocks of source code with a single identifier, are used, a separate file is maintained which contains each common-deck identifier and a list of all modules which utilize the common-deck. Assembly language macros are handled in the same manner.

(2) Object Library

The object library contains target computer code, usually in relocatable form, which has been produced by a translation of the source code. If a particular software effort uses both a high-level programming language and assembly language, then two distinct object files are maintained in the FASP. The object files are retained in formats which are compatible with the respective linkage editor products.

(3) Test Library

The FASP employs the concept of regression-testing wherein sets of test input data and test results are accumulated during the development of a software system. The test input data is provided by the software engineer who is responsible for testing the software module; the test results are obtained from a target computer software-emulator which executes on the host computer. In addition to test data, test results and test directives, an index of what tests are to be performed for each software module is maintained. When modules are changed, the FASP automatically repeats the test and compares the new results to the old, giving appropriate printouts if discrepancies occur.

(4) Interface Data Library

It is standard practice to organize large software programs in such a way that common data can be shared by many modules; similarly, it is natural to develop general purpose routines which can be called from several modules. When a module executes a call to a separate routine or procedure, the computer transfers control to an entry-point of the called routine; sometimes a routine may have several entry points. The term "entry point", or sometimes simply entry, is also used with variables to denote where the variable is assigned storage.

In the FASP, a file is maintained of all external references and entry points. The information is obtained by a FASP processor which scans the actual object files. A FASP user can therefore obtain a list of all modules which reference a particular external variable name as well as all associated entry points for the external variable.

(5) Production Data Library

This library contains information such as modification histories and management information. Management information is captured down to the module level for each data base; the information includes module name, module size in source and object, creation date, date last modified, programming language, number of test runs, etc. For certain parameters estimated values are stored in the data base and can only be changed by project managers; FASP then tracks estimated values versus actual and gives on-line reports of the differences. Some of the estimated parameters are number of modules, number of source lines, number of object words, etc.

FASP System Software

From figure 3 it can be seen that all interaction with the user is handled by a command processor. The command processor invokes a FASP procedure which in turn calls on other software to complete the procedure; this latter software is categorized as support packages, software engineering tools, and software management tools. The interaction with the project data base is handled by the FASP Data Base Management System (DBMS).

While FASP procedures rely on system software to perform tasks, FASP is function-oriented rather than tool-oriented, differing in this respect from other software generation facilities. In a tool-oriented facility, the user is responsible for calling each tool separately, supplying all the detailed parameters, providing the host computer's job control language, and insuring that the interface data to all other tools is correct. With FASP, the user specifies the function in simple terms and FASP supplies the appropriate tool or set of tools in proper sequence.

FASP Procedures

FASP provides a set of user commands to accomplish all the software generation functions that must be performed by programmers and managers during the various phases of the software life cycle. FASP procedures have been designed both to simplify the programmer's task and to enforce standard modes of development on all users.

A procedure is invoked by issuing a FASP command. A command consists of a directive indicating the overall task to be performed along with the required parameters and data. The associated procedure may perform many smaller tasks to carry out a command; many procedures are designed to carry out standard programming sequences. For example, one FASP command specifies "software modification". The associated procedure updates source code, translates it, stores the object code, updates the interface and production data, and at the user's option will perform testing and compare results to those of previous tests. This grouping of well-defined programming sequences adds another dimension of standardization to the software production process.

Through the use of FASP procedures, all programmers follow the same development pattern and are, therefore, able to understand the software developed by others as well as being able to easily shift their efforts to other projects using FASP. In addition, FASP handles all the details required for using system software to accomplish a task. These details include selection of options, interfacing different formats, and specifying the sequence in which tools are invoked when more than one is needed for a given task. Examples of the tasks that are supported by FASP user procedures are:

- Data Base Functions
  - data base creation and deletion
  - data base saving and restoring
  - data base auditing
- Software Functions
  - software creation and deletion
  - software translation
  - software editing and modification
  - software sharing and copying
  - software listings
- Load Tape Generation
  - load module creation
  - load tape creation
  - load tape transmission
- Test Functions
  - test creation and deletion
  - test execution
  - test modification
  - regression tests
  - automated test analysis
  - test listings

Support Packages

This category of FASP system software contains the common support software products which are necessary to generate weapon system software. There are three sub-categories: Editors/Librarians, Translators/Preprocessors, and System Generators.

Editors/Librarians

This type of software includes interactive text editors and programs called librarians which maintain source files, object files, and application libraries. The FASP interactive text editor has all the features found in modern text editors and is compatible with the source program librarian. Some of the features are: each source image is given a unique identifier, each source image is kept in active or inactive status; a chronological history is kept of changes to the status; common units of code can be inserted anywhere; correction sets may be identified, inserted, or deleted; the source program is stored in a compressed format; simple commands exist to ADD, DELETE, or MODIFY the source program.

Translators/Preprocessors

The term translator is used to denote FASP processor which translates a programming language source program into a target machine object language in either relocatable or absolute form. Translators include high-level language compilers, assemblers, and microprogram translators (sometimes called microassemblers). The present environment is such that weapon systems contain several different computers; thus, several translators must be available in FASP. Further, it is not uncommon to have different translators each at a different release-level. Modern compilers usually contain separable code generation sections in order to accommodate different target computers; however, within the FASP the code generators are considered bound to the main compiler and the combination is treated as a single translator. A goal is to have the interface with the translators as uniform as possible and, thus, to avoid any reformatting between the source library and translator and also between the translator and object library. A requirement on all translators is that the name and version number of the translator be stored in the object library for each translator run.

A preprocessor or precompiler is a program which operates on the source program prior to translation. It is used to provide features not existing in the translator proper, for example, a conditional translation option, a macro-type facility for a high-order language, or the implementation of structured programming constructs.

System Generators

The term system generator is generic and refers to a collection of products which operate on data in the object library and produce load tapes for the target computer. These load tapes are usually in absolute format and contain the operating system (or executive) and application programs for the target computer; additionally, the load tapes may contain microcode for the target computer, hardware diagnostics as well as complete sets of object code for other target computers in the weapon system (the case where the first target computer is loading the other target computers). Included in the system generator category are linkage editors and relocating loaders; the essential characteristic of these products is flexibility in searching various object libraries and quickly creating the correct load tapes.

Software Engineering Tools

The term "software engineering tool" covers a variety of programs which assist the software engineer in developing and maintaining weapon system software. Two of the most important FASP tools are discussed below.

Software Emulators

The term software emulation refers to the process where a computer is imitated by a program such that the imitating program accepts the same data, executes the same object code, and achieves the same results as the actual computer. This is in contrast to a hardware emulation of one machine by another, for example, through microcode. Software emulators can be relatively quickly implemented for existing military target computers or for new computers once the instruction set and architecture are firm. Software emulators are typically implemented with either interactive control from a conversational terminal or with trace-command controls in a batch mode. One of these latter features is essential because software emulators are typically several hundred times slower than execution on the actual target machine; the control features allow emulated execution of relatively short segments of code, which is quite adequate for debugging runs during the software development. However, since software emulators do not run at the same speed as the actual target computers, computations are usually performed within the emulator to produce timing approximations within a few percent. The principal advantage of software emulation is that software development can begin early in the overall development cycle; software testing can begin before delivery of the actual hardware and, thus, give the software engineers considerable insight into the adequacy of their designs. In using software emulation, one must insure that the emulation maintains a one-to-one correspondence with the actual hardware; however, since the emulation is performed at the instruction level, this is not a difficult problem.

Some additional advantages of software emulation in FASP are:

- Many users can simultaneously checkout software rather than sequentially using actual military computers.
- The user has access to the emulated machine at the high-level of the emulation language, typically FORTRAN, which allows the contents of memory, registers, and status lines to be obtained in a convenient format, e.g., decimal notation vice hexadecimal or octal, without disturbing the program under test.
- Within the emulation, extensive error detection can be implemented which may, in fact, go undetected on the actual machine, e.g., addressing beyond the bounds of available memory, or improper double operand alignment.
- The emulated machine may be used to evaluate hardware modifications before they are made in the actual hardware, or certain modifications may be made to facilitate software checkout and then removed before the software is run on the actual hardware.

Automated Test Analyzer (ATA)

The ATA automatically scans the source program, determines the paths between decision points, and instruments the source code without altering the intended computations. The instrumented program is then dynamically executed with test data while special run-time routines record data; the dynamic execution may be on the software emulator or the actual hardware. The recorded data is post-processed to show what paths have been tested as well as the frequency of execution. The resultant statistics are accumulated over many tests, forming a complete test picture. A project manager can assess the progress of the testing effort and has a quantitative indication of the risk associated with releasing software which is not 100% tested. In this case 100% testing means all statements have been executed at least once. The frequency of execution of certain paths provides information to the software engineer for improving the performance of the software.

SOFTWARE MANAGEMENT TOOLS

Software management tools refers to the collection of programs which produce software management information from the project data base. The most important tools in this category produce the FASP software management reports; these reports are module-detail, module-summary, data base-detail, data base summary, and account summary. The account summary report contains information on all data bases assigned to the particular account number; an account number is the identifier used for host computer cost accounting. This allows the convenient collection of technical data and cost data on any project software effort.

Extensive reports are produced concerning the host computer resources. Items such as number of runs, CPU time, Turnaround Time (TAT), etc., are monitored on a weekly basis. Also reported are details of the FASP operations, such as what procedures were used, number of load tapes generated, etc.

FASP OPERATION

The purpose of this section is to provide the reader with some insight into how the integrated FASP components form a programming system. Figure 4 shows a view of FASP which is functionally oriented. The user interacts with a FASP software processor which has access to the FASP system software and the project data base as previously described. In this view the FASP processor produces either test outputs or a tape; the tape is called a load tape and is the means for transferring the software to the target computer. The test outputs are normally hardcopy results of compilations or simulator runs.

A functional view of the FASP processor is shown in figure 5; five major steps are shown – editing, translating, linking/loading, and either simulator testing or load tape generation. If we retain this functional view and add a view of the project data base we obtain figure 6. A typical scenario is as follows: first, an existing module (OLD SOURCE) is corrected or modified in the editing phase producing a new module (NEW SOURCE). Second, the revised module is translated by a compiler or assembler producing a new object file; the interface data is updated by routines in the FASP procedure. Next, the object code is bound together producing a load module. If

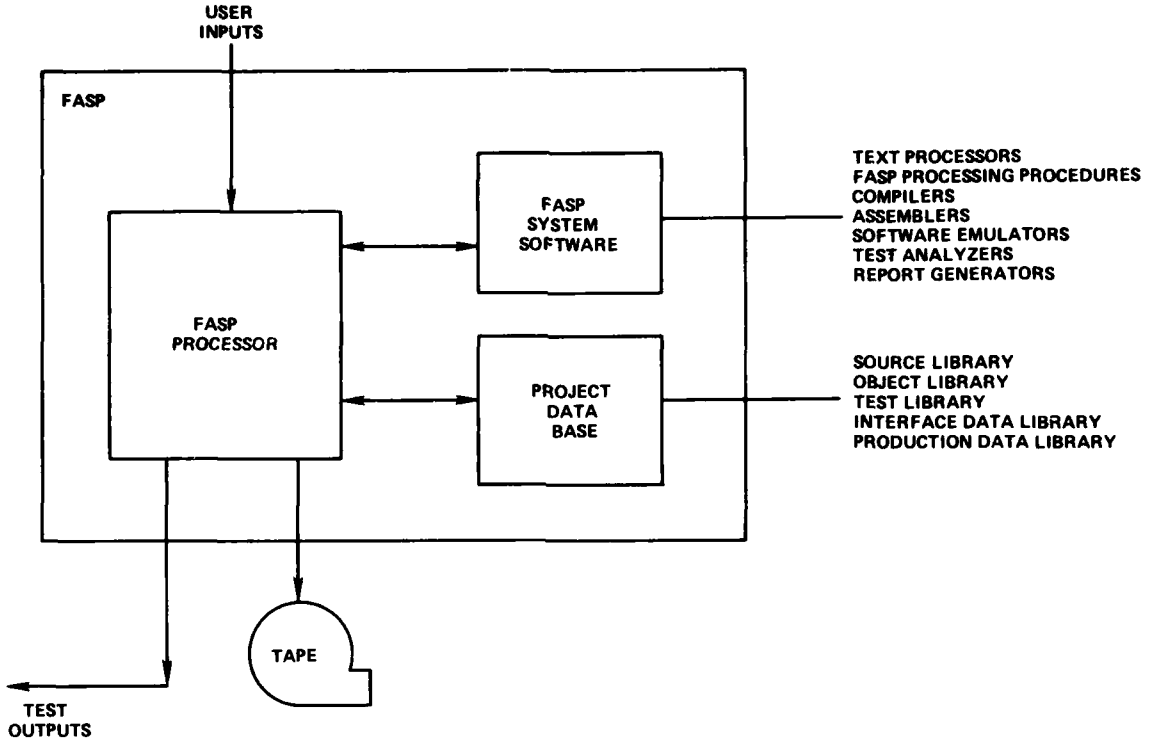


FIGURE 4 – Facility for Automated Software Production

the software is to be executed on the target computer a load tape is generated in the particular format of the target computer; frequently, the physical magnetic tape is nonstandard for military computers. If a simulator test is specified the test data and directives are used by the software emulator which in turn produces the test results. Information is continually added to the production data file during all the operations. References 11 through 14 describe FASP in greater detail.

Regression Testing

An important feature of FASP is regression testing. During software module development all test data, inputs, outputs, and test directives, are stored in the data base. A full test history is thereby established for each module. Figure 7 shows the flow during regression testing. Once a module is developed, any change will trigger automatic rerunning of all test cases with a printout of any differences between new and previous results.

For each data base a test directory is maintained which specifies a set of tests for each module; when a module is changed the set of tests is repeated and for each test the new results and old results are compared.

Regression testing is most important during the maintenance phase when it is critical that new changes do not cause software errors in unintended functions.

5. FASP EXPERIENCE

The purpose of this section is to relate the NADC experience in implementing and using FASP. The implementation experience is important because of the very fundamental tradeoffs which were made.

IMPLEMENTATION

Distributed Network Versus Central Site

When implementing a system such as FASP a very crucial choice exists at the outset as to the configuration of the host computer(s); the issue is not remote access because that can be solved by a communication network regardless of the host computer configuration. The choice is whether the project data base and all system software will be at one site or will the resources be distributed across several sites such that parts of the software development can be done at any site. Conceptually, either approach is feasible; however, there are some serious practical problems with the distributed approach.

The basic problem centers around the project data base; in the FASP concept the various libraries in the data base have very specific dependencies on one another. For example, the source and object libraries must be kept in one-to-one correspondence, the histories of changes must be accurate and the management information must relate to the total data base. Another difficulty is the recovery of the total data base in the event of loss or damage to a particular library.

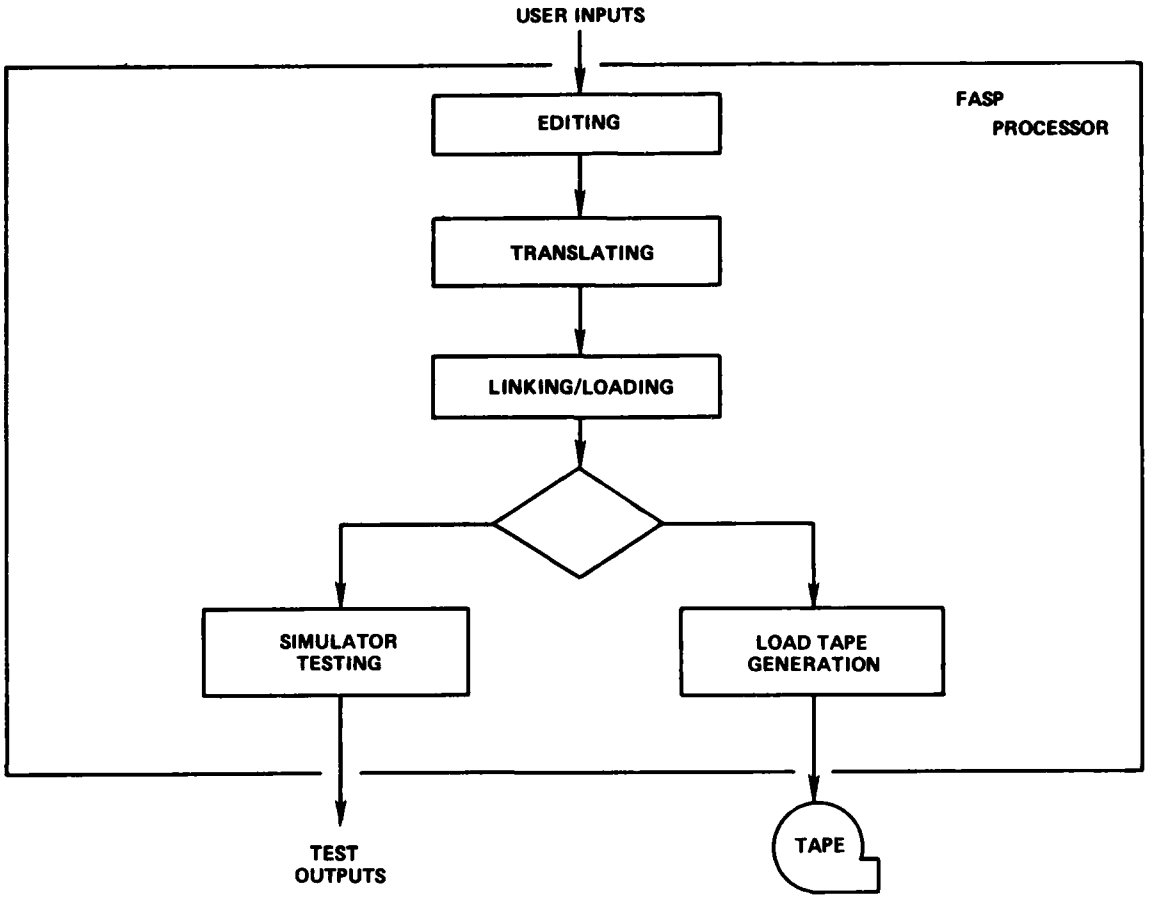


FIGURE 5 – Development Steps

A further difficulty in using a distributed approach is the compatibility of all the subcomponents of the FASP system software, e.g., compilers, emulators, etc.; this is a formidable problem especially when different host computers are used.

In the case of FASP the implementation was chosen to be at a central site with remote access by available communication networks. It was intended that interface-control-documents would allow the easy exchange of source programs and object code with other facilities.

Host Computer

The choice of a host computer is extremely critical. It was clear that a commercial computer rather than a military computer should be used; the great variety of peripheral equipment and the large software base was far superior to that of military computers. The issues became choosing the size of the host computer and the management factor of dedicated operation to a single project or multiproject support.

Since it was a strategy with FASP to trade computer-time for labor-time and to apply more tools during the software development, a large scale computer was desired. On the other hand, the cost of a large scale computer would be difficult to justify for a single project.

At NADC there existed a large scale centralized computer complex consisting of two CDC 6600's which could be further expended if needed. This facility was selected as the host computer.

Project managers were concerned about the cost of operation and the availability of computer service; factors which were under total project control with dedicated facilities. These concerns were solved when by policy the computer costs were stabilized for a five-year period and firm, written guarantees were given for host computer turnaround time (TAT).

Today the NADC computer facility has two CDC 6600's, a CDC CYBER 170/175, and a second CYBER is planned to be added in 1980. These computers operate independently but all share a common file system; the same version of the operating system executes on each computer. This multicomputer installation is a tightly-coupled distributed processing configuration with high performance for the user.

Data Base

The use of a project data base, managed as a whole rather than as a dispersed set of files, is a critical part of FASP. The data base and the underlying relationships between the libraries was the unifying element in the design and implementation of FASP. One study

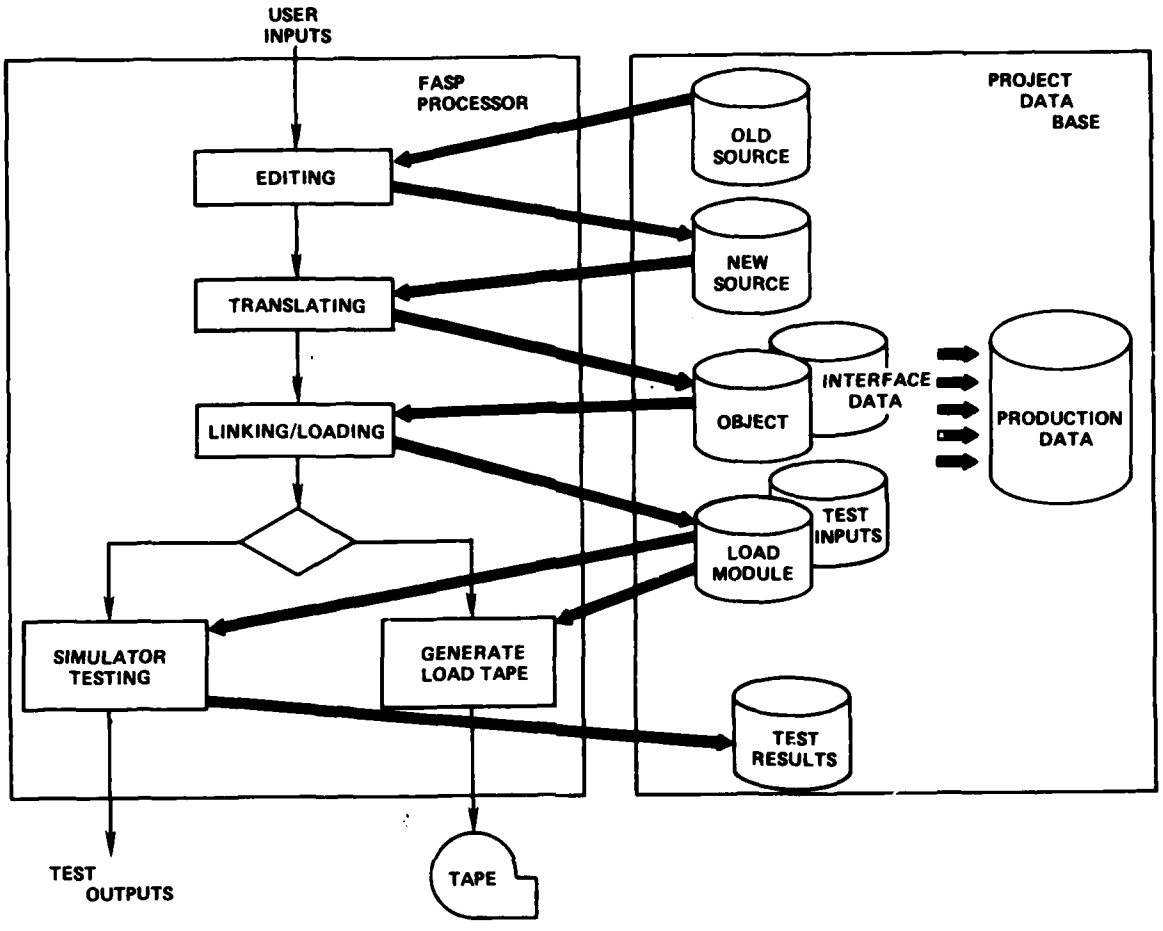


FIGURE 6 – Data Base Interaction

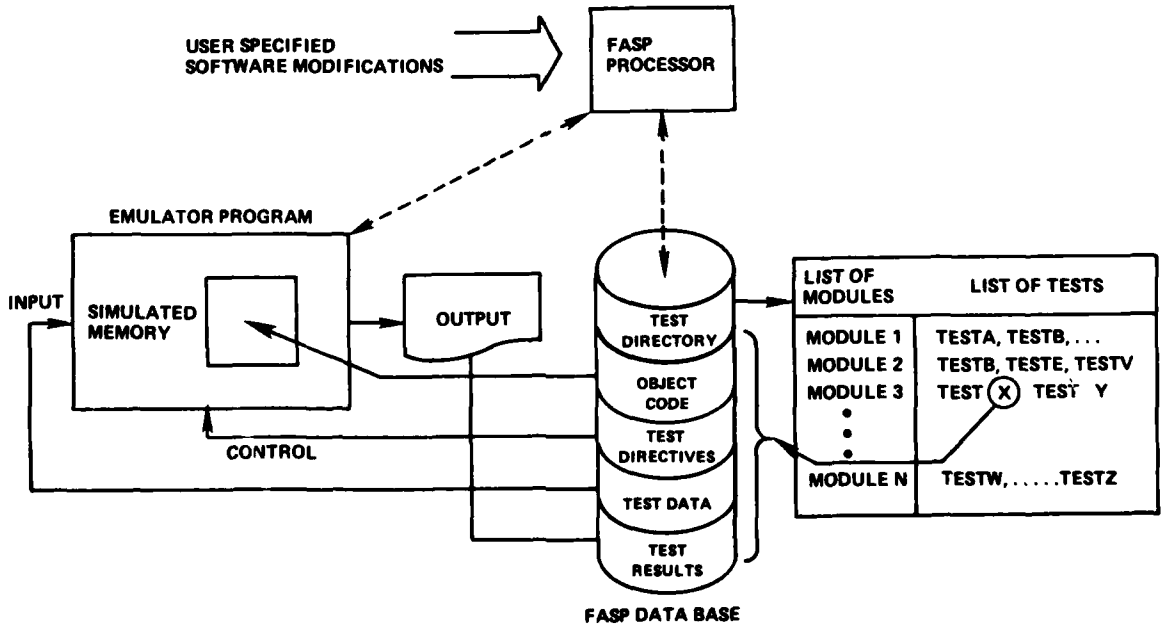


FIGURE 7 – Regression Testing with FASP

recommended the development of a new DBMS and a new access language which also would serve as a job control language (see references 9 and 10). A survey of existing DBMS's showed that no existing system could meet the requirements; some of the factors were:

- the ability to handle long-sequential strings of data as found in source programs, binary object files, etc.
- most DBMS's used inverted file structures, a feature which would tend to enlarge the total size of the data base for small to moderate amounts of data.
- the project data base was an application where the "sort keys" were known and, therefore, a general approach to selecting data elements was not required.
- some of the contents of the data base had to come from compilers, assemblers, etc., a difficult problem with available DBMS's.
- most DBMS's did not allow multiple access to a single data base nor did they have adequate security controls.

Rather than develop a new generalized DBMS a specialized FASP DBMS was developed using the standard file system of the host computer. This approach has the disadvantage in that it is difficult to add a new type of library; however, it has the advantage of being very efficient when running and straightforward to implement.

Interactive Versus Batch

This topic has been of considerable debate. There has been, and continues to be, a growing body of evidence that clearly shows that a programmer's productivity is increased with the use of interactive capabilities. There is also evidence that programming teams which employ program librarians can achieve high productivity with batch systems. It is sometimes argued that interactive capabilities give too much freedom to the programmer, that extra computer runs will be made, that the work will be hurried, careless, etc.; whereas, the thoughtful skilled programmer who carefully analyzes each step will more carefully use a batch system yielding higher quality software at an overall higher productivity. This is a weak argument since interactive capabilities in the hands of thoughtful skilled programmers would clearly solve the issue.

In the case of FASP, the limitations of the original operating system (SCOPE 3.3) were such that interactive capabilities were impractical. A change in operating systems (KRONOS 2.1) made it feasible to introduce a degree of interactive capability.

From a functional standpoint, (figure 5), two points were selected for interactive features – editing and simulator testing. Both have been implemented while retaining batch operations. It is not planned to convert the existing FASP system software to be fully interactive, e.g., incremental compilers; however, all new components are planned to be highly interactive.

PROJECT EXPERIENCE

FASP Usage

The total FASP usage in terms of computer jobs per month is shown in figure 8. At any given time the number of projects using FASP averages about 20; about 6 are large scale efforts. FASP accounts for about 1/3 of the total jobs on the host facility and about 2/3 of the resources, i.e., CPU time, file space, etc. FASP supports 6 different target computers. In general, the larger projects support the development of a FASP; however, the smaller projects can use these capabilities, paying only the computer usage costs.

Since the FASP procedures are highly uniform across the target computers, both in-house and contractor personnel have been able to be moved from project to project with relatively high efficiency.

The total FASP usage presents a considerable workload to the host facility; figure 8 also shows the average TAT for the FASP workload. As projects neared critical deadlines the computer utilization, e.g., CPU time, increased dramatically. The large-scale computers were able to handle this peak workload and keep the projects close to schedule; however, the large workload caused the scheduling algorithms to thrash, increasing TAT. A contributing factor was inefficiencies in the FASP system software, in particular, the compilers, assemblers, and software emulators. After analysis the combination of performance improvements and modifications of the scheduling algorithms solved the TAT problem. The large-scale computer with the increased TAT basically caused multiple shift operations but accomplished the work; smaller computers would have become so saturated as to result in lengthy schedule delays.

Figure 9 shows the number of jobs per month for two representative FASP projects.

Software First

The use of a software emulator, or simulator, to develop and test the software prior to execution on the target computer was a controversial issue. The major weapon system contractors were divided on the issue; some stated the project could not be done without a simulator, others that none was needed. In the end all of the projects agreed it was a very valuable tool. Late deliveries of the target computers and access to the integration facilities caused very difficult scheduling problems; the simulators on the host computer proved a viable alternative. After using the simulator the testing in the integration facility was significantly quicker. Getting the software errors corrected early in the schedule was not only cheaper but also yielded higher quality software.

Software Estimates

For contractors who were new users of FASP a comparison between their proposals and subsequent actual usage showed large discrepancies. The differences were in all areas but most noticeably in the total amount of software developed and the rate at which they could produce software; thus, the total job was underestimated in size and the team was generally not able to produce software at the expected rate until late in the schedule. With the feedback of information from FASP the differences between proposed or estimated values and actual values rapidly decreased.

In the case of dedicated small scale facilities the same discrepancies occurred but there was no corrective feedback path.

Productivity

Productivity, in the most general sense, is an important economic indicator; it is the ratio of what-is-produced to what-was-required-to-produce-it. Unfortunately, it is not a pure independent variable and factors such as the quality of the product are not directly

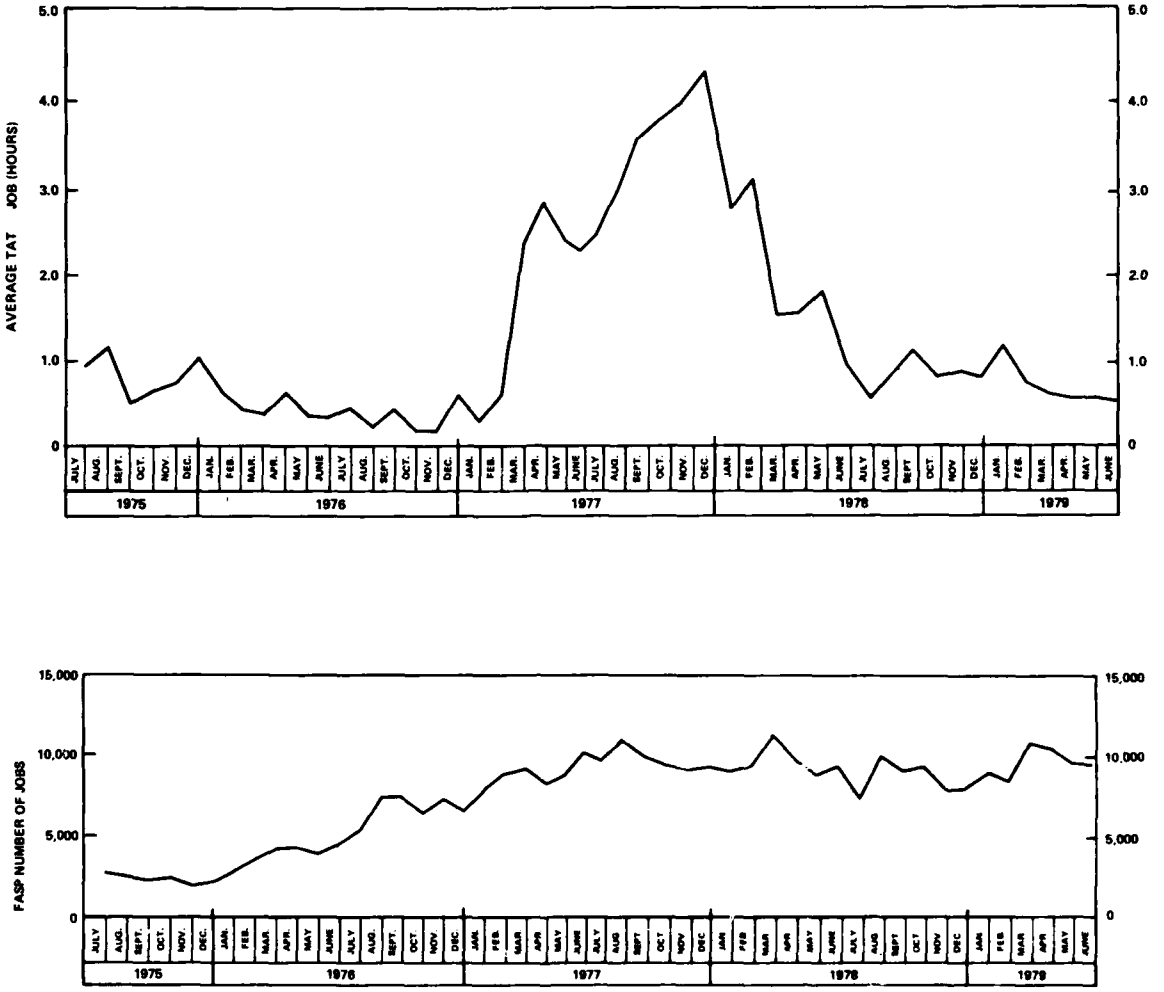


FIGURE 8 – Average TAT for the FASP Workload

included. Software, like many other areas, is labor intensive; that is, labor costs dominate. As in other fields, technological advances must increase the productivity of labor or costs will spiral upward indefinitely. With FASP a goal has been to increase productivity; however, it is difficult to prove this point in an absolute sense. To do so would require doing the same project with and without FASP while holding all other factors equal, a nearly impossible task.

At NADC two measures of productivity are tracked, Delivered Source Lines, including comments, per Man-Month (DSL/MM) and Delivered Object Words, including program and data, per Man-Month (DOW/MM). It is assumed that the amount of testing is relatively high and that full documentation is included. Representative samples of productivity with FASP are shown in table I; note that all projects are real-time weapon system applications. A note of caution with regard to the data of table I; the complexity of the application is a critical factor. Project C was significantly more complex than the others. Overall, the productivity with FASP was considerably better than with smaller scale dedicated facilities.

6. CONCLUSIONS

FASP has demonstrated improvements in the software problem areas of life-cycle cost, quality, and delivery-schedule. Providing a facility with the dual functions of an advanced programming system and a management information system has been a significant factor in the acceptance of FASP. The underlying issue is that by standardizing and stabilizing the programming environment, many of the old software problems have disappeared.

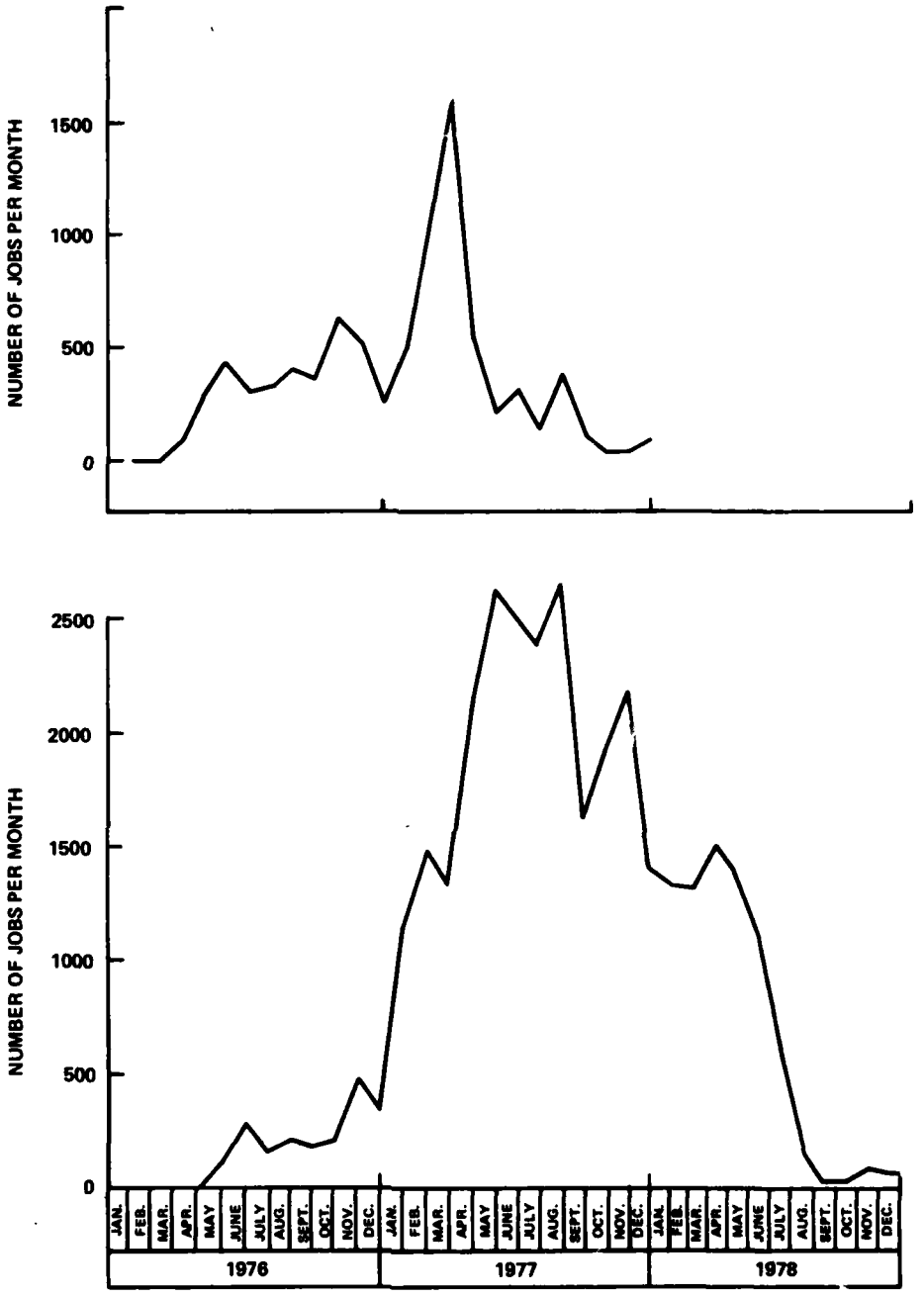


FIGURE 9 – Total FASP Usage in Terms of Computer Jobs Per Month

TABLE I – REPRESENTATIVE SAMPLES OF PRODUCTIVITY WITH FASP

		Language	DSL	DSL/MM	DOW	DOW/MM
Project	A	AL	95,500	424	65,000	286
	B	AL	108,000	521	65,000	313
	C	HOL	1,600	58	5,000	185
	D	HOL	12,200	420	16,600	573

AL – Assembly Language
HOL – High-Order Language
MM – Man-Months (176 hours/month)

Referring to figure 1, it would be ideal to have highly automated facilities to support all the activities, but what is the best way to reach that goal? Requirements and design methodologies with automated support tools are judged to be in a largely experimental state. Therefore, selecting the code and test activities was key to making basic changes in the way weapon system software is developed and maintained. The effort was larger than anticipated because not only was a significant technical development required but also a basic change in the way the managers and programmers accomplished their work. However, once the code and test activities were accomplished the necessary changes to use requirements and design tools are expected to be small.

On the average the initial programmer reaction to FASP was negative; however, this changed dramatically when it became apparent that FASP was merely itself a tool. Their creative talents could be focused on the end product rather than support tools, a result which improved the satisfaction of the end user. FASP produced the welcome benefit of making the managers not only more knowledgeable of their product but also more appreciative of the actual efforts of their programming staffs.

7. REFERENCES

1. CCIP-85. Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's: Executive Summary, Revised Edition. Los Angeles, CA; Air Force Systems Command, Space and Missile Systems Organization, February 1972. (AD 742 292)

   CCIP-85. Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's: Eleven Volumes. Los Angeles, CA; Air Force Systems Command, Space and Missile Systems Organization.

Volume I	Highlights	(AD 900 031L)
Volume II	Command and Control Requirements: Overview	(AD 521 887L)
Volume III	Command and Control Requirements: Intelligence	(AD 523 881L)
Volume IV	Technology Trends: Software	(AD 919 367L)
Volume V	Technology Trends: Hardware	(AD 907 626)
Volume VI	Technology Trends: Sensors	(AD 525 661)
Volume VII	Technology Trends: Integrated Design	(AD 906 757L)
Volume VIII	Interservice Coordination Trends	(AD 522 216L)
Volume IX	Analysis	(AD 524 549)
Volume X	Current Research and Development	(AD 905 654L)
Volume XI	Integrated Research and Development Roadmaps	(AD 902 515L)

2. Goldberg, Jack, ed. The High Cost of Software, (Proceedings of a Symposium). Sponsored by: Air Force Office of Scientific Research, Army Research Office, and Office of Naval Research, Monterey, CA; 17-19 September 1973. Published: Menlo Park, CA; Stanford Research Institute. n.d. (Contract N00014-74-C-0028)

3. Project PACER FLASH. Four Volumes. Wright-Patterson Air Force Base, Dayton, Ohio; Air Force Logistics Command, 28 September 1973.

Volume I	Executive Summary and Final Report
Volume II	Appendix A: Automatic Test Equipment (ATE)
Volume III	Appendix B: Operational Flight Program
Volume IV	Appendix C: Air Crew Trainers (Simulators)

4. Reich, Eli T. Tactical Computer Software Acquisition and Maintenance Staff Study. Washington, D.C.; Deputy Assistant Secretary of Defense, Production Engineering and Material Acquisition, 31 October 1973.

5. Electronics-X: A Study of Military Electronics with Particular Reference to Cost and Reliability. Arlington, Virginia; Institute for Defense Analyses, Science and Technology Division; January 1974. (R-195)

6. Fisher, David A., Automatic Data Processing in the Defense Department. Arlington, Virginia; Institute for Defense Analyses, Science and Technology Division; October 1974. (P-1046) (Contract DAHC15-C-0200, Task T-36)

7. Asch, A.; Kelliher, D. W.; Locher, J. P.; Connors, T. The Mitre Corporation, DOD Weapon Systems Software Acquisition and Management Study, Volume I, Findings and Recommendations, MTR-6908, May 1975.

8. Kossiakoff, A.; Sleight, T. P.; Prettyman, E. C.; Park, J. M.; Hozan, P. L.; The Johns Hopkins University Applied Physics Laboratory, DOD Weapon Systems Management Study, APL/JHU SR 75-3, June 1975.

9. Softech, Inc.; Support Software Planning Study; Contract N62269-74-C-0269, March 1974; U.S. Naval Air Development Center, Warminster, PA.

10. Irvine, C. A.; Brackett, J. W.; Automated Software Engineering Through Structured Data Management; IEEE Transactions on Software Engineering, Volume SE-3, No. 1, January 1977.

11. FASP Brochure – U.S. Naval Air Development Center, Warminster, PA, 1 May 1978.

12. FASP Management Summary – U.S. Naval Air Development Center, Warminster, PA, 13 April 1979.

13. FASP Software Production and Maintenance Methodology – U.S. Naval Air Development Center, Warminster, PA, 10 July 1979.

14. FASP Handbook – U.S. Naval Air Development Center, Warminster, PA, December 1979.

LOGIC STRUCTURE FOR TESTABILITY AND FAILURE DETECTION

U. Schulz and A. Roelker

Dornier AG
Postfach 1360
7990 Friedrichshafen
West Germany

ABSTRACT

A concept of the logic structure for testability and failure detection of digital, modular, onboard data acquisition and control units within avionics systems will be presented. The units of the avionics system are interconnected by the serial Data Bus (1553).

The concept was developed according to the maintenance levels within the German Air Force. The tests and failure detection are performed during operation as well as during pre- and post flight tests without additional external special to type test equipments. The tests are totaly integrated into the avionic system.

The tests result in the localization of the failure on the defective module within a unit. The test-software needs about 4 msec and may therefore incorporated into the operational software.

This leads during operation (inflight) to a reorganization of the system by software such that the full system capability will be kept until a second failure occurs.

This system capacity is performed by a minimum of additional hardware (5%) and testsoftware (10%).

1. INTRODUCTION

The conventional structure of onboard electronic and avionic system is purely a collection of units and subsystems (for example: navigation, guidance and control, communication, displays, ...) which are separated and independent from each other. The failure detection and testing of such systems is performed by many special to type testequipments for pre- and postflight tests as well as for maintenance tests. Testing during operation is nearly impossible except the monitoring of a failure of a complete subsystem. A concept for the testability and failure detection of integrated, digital onboard electronics within avionics systems is presented. The system makes use of the inherent redundancy.

2. ASSUMPTIONS

The following assumptions are made:

- The avionics system is structured as an integrated data processing system with a uniform structure concerning the signal processing soft- and hardware.
- The system contains partially redundant units due to the specified reliability of the subsystems.
- The signal processing units are line replaceable units (LRU) and interconnected by the serial data bus according to MIL-STD 1553.
- The internal structure of the LRU's is modular with the modules connected and operated via a parallel data bus. The modules are functional electronic boards as multiplexer, A/D-converters, digital inputs and outputs, ...
- Most of the LRU's are controlled by an internal processor.

The following approach on testability and failure detection is based on the system shown in figure 1. This is an example of an avionic system design for a helicopter for which the previous mentioned assumptions are valid.

3. CONSIDERATION OF THE PROBLEM

The failure detection and testing is divided into two main levels, according to the so called maintenance levels of the German Air Force, figure 2:

- o Testlevel 1:

  During the testlevel 1 the inflight test and the pre- and postflight test will be performed. The task of the tests are to detect the failed function of the system during operation: System GO/NOGO.

- o Testlevel 2:

  The testlevel 2 is the first maintenance level. The tasks of the tests are to detect the failed hardware components.

The failure detection during operation (test level 1) is performed by well known methods. Those methods are the observation of the process, the comparison of redundant sensor-signals, the voting of the results form the algorithms within redundant data processing units.

Therefore the following considerations are at first limited to the maintenance tests. (test level 2).

4. HARDWARE TESTABILITY

The maintenance test has to lead to the detection of the failed hardware components. The considered system (see figure 1) has a uniform structure of the signal processing soft- and hardware, with LRU's built up out of standard modules.
Figure 3 shows the general configuration of a modular LRU with an electronically and mechanically standardized parallel data bus for the internal data transfers and the serial bus coupling module for the external connections to the other units. The units contain further a data processor, a memory, RAM or PROM, some autonomous modules that are only needed for the control of the internal events, and a lot of digital and analog I/O-modules that perform the connection to the process.

If we want to test such a system we must have testable hardware. Fig. 4 shows the structure of a testable analog peripherie. We get the testability by using the inherent redundancy of the modules. The n- and m-input signals are given into the both input modules and are given back again to the system by additional wiring from the outputs to the inputs (cross strapping). The test itself than is performed by software such that a failure within the analog peripherie may be detected and localized down to the level of analog input or output modules.

In order to test the digital periphery the information is given twice to the processor on different ways and thus providing information redundancy.

The figure 5 shows the structure of a testable digital I/O-module. Here some circuits are added to the pure module function of every module in order to set up the test information and to send it on a "digital test bus". The test information is converted to a serial 4-bit signal.

The figure 6 shows the receiver module that is connected to the parallel memory bus of the data processor. The test information than is transfered directly to the registers of the data processor and there compared against the measurement value that comes directly over the data bus from the module.

The additonal hardware for the test of the digital periphery consists only in one receiver module, the circuits on the electronic boards and the wiring of the serial test bus. The tests itselves are made by software.

Figure 7 shows an example of a realized LRU within an experimental flight guidance and control system. The LRU is prepared for the maintenance test.

The analog periphery is tested as described before with the additional wiring (cross strapping) as shown in figure 4. The test of the analog periphery includes also the signal conditioning or filtering modules.

The digital input and the parallel data bus are tested by means of an additional module, the data way memory. This is in addition to the previous mentioned way another possibility to test the digital periphery together with the parallel data bus.

A failure of a bus line for example is detected by transmitting a testsignal through the bus to the register of the data way memory. From there it is transmitted back to the processor at one way through the bus and at a second way via the process-side and the digital input. The wiring between data way module and digital input is performed such that the first 8 bits of the test signal are given back to the processor through the last 8 bit bus lines and vice versa. If there is a failure for example of a bus line, the two testsignals transmitted via different ways are different.

The test of the LRU consists further of short tests of autonomous modules (clock, ...), the data transfer via the serial bus, the processor itself, the memory and power modules inside the LRU.

From investigation and tests we got the following results.

Considering the results of all tests and their logical combination the test leads to a localization of the defective module with a LRU.

The calculation of such a test takes about 500 command words and a time of about 4 msec.

This leads to the possibility to perform the test at every calculation cycle of the processor while the system is full operatin.

Due to this capability of failure localization at the level of the module during the operational mode it is possible to reorganize the software of a LRU after the detection of a failed module. The information for example coming from an analog input may be replaced in case of a failure of the module by a value that is calculated from redundant informations within the system (additional observer).

Figure 8 now shows the block diagram of the software structure of a LRU. The test software now is incorporated into the operational software. The organisation and timing of the complete software is performed by a task scheduler.

After the discussion of the testability of one LRU the maintenance test of the whole system (see figure 1) is considered.

In principle there are two possiblilites to realize the maintenance test:

1. The test will be performed by a simple external test terminal with the test software situated in the memory of every LRU and the task scheduler in the terminal.

2. The test will be performed by internal test procedures that are situated completely within the memories of the LRU's and performed and controlled by one LRU being the master on the serial data bus during the test phase.

An example for the configuration of an external test terminal is shown in the figure 9. The whole system is connected to the control and test terminal (CTT) via the standard interface of the serial data bus. With this configuration the whole system may be tested with the CTT being the master on the bus. On principle the same configuration is valid if the test will be performed by internal test procedures. In this case the communication between the system under test and the operator may be performed via a multifunction display and a multifunction keyboard in the cockpit (see figure 1).

With respect to the new function of the serial Data Bus according to MIL-STD 1553 B "Transmit BIT-Word" and the above described possibilities for the test and failure detection of LRU's that are structured modulare there are the following results:

Inflight-, pre- and postflight-test as well as the maintenance test according to the testlevels 1 and 2 may be performed and totaly integrated into the system.

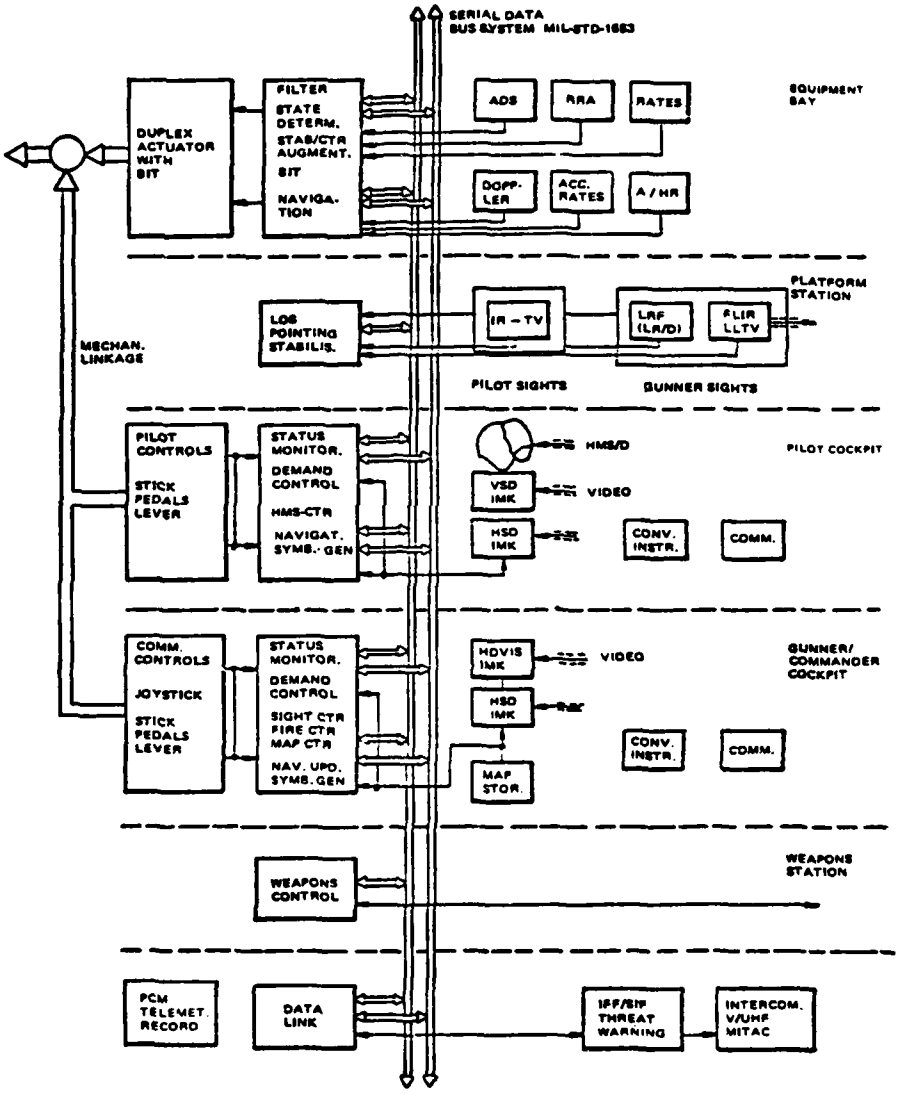
There is the possibility to detect the failed function of the system and to localize the failure at the level of the module within the LRU.

There is no need of any external test equipment if the result of the failure detection and the test may be indicated (immediatiely or on request) on a multifunction display in the cockpit. The decision whether the information of the failure must be indicated immediately or on request during the flight depends upon the importance of the failure and its influence at the operational mission. During pre-, post-flight and maintenance test all results might be indicated.

With respect to the testable hardware and the methods of failure detection within redundant systems it is possible to allow one detected failure without degradation during operation (inflight). The software of the failed LRU will be reorganized as discribed above. The full system capatibility will be kept until a second failure occurs.

This is especially interesting for example in the case of a triplex guidance and control system.

This capacity of the system is performed by a minimum of hardware (5%) and the test software (10%) incorporated into the operational software program of the system.

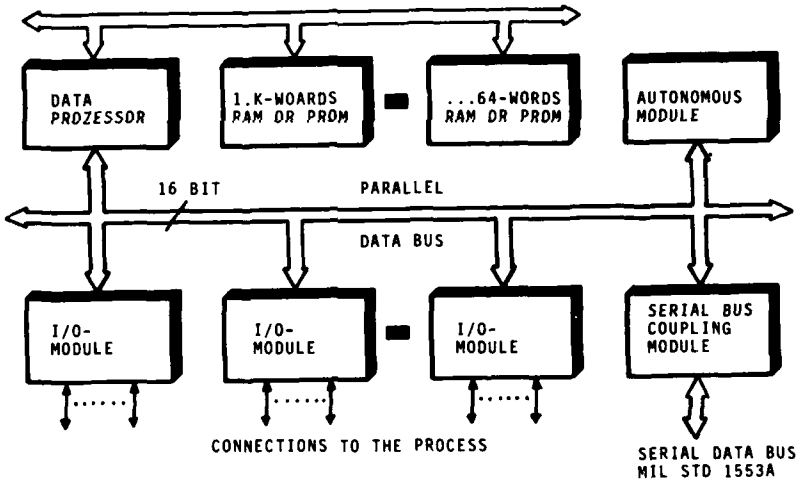


DORNIER

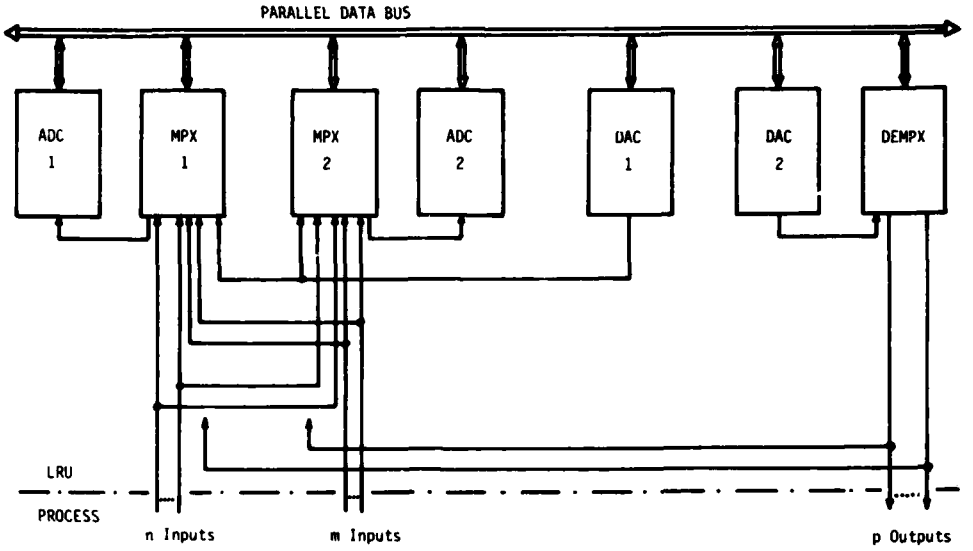
FIG. 1: **AVIONIC SYSTEM DESIGN ALTERNATIVE: MECHANICAL BACK UP**

TESTLEVEL 1	TESTLEVEL 2
o DETECTION OF A FAILED FUNCTION OF THE SYSTEM	o DETECTION AND LOCALIZATION OF THE FAILED HARDWARE COMPONENTS
OPERATIONAL MODE	SUSPENDED OPERATIONAL MODE
LIMITED TIME	NO TIME RESTRICTIONS
THE SYSTEM MUST BE CONTROLLED BY THE PROCESS SOFTWARE	THE SYSTEM IS IN A STABLE CONDITION
INFLIGHT PRE/POST-FLIGHT	MAINTENANCE

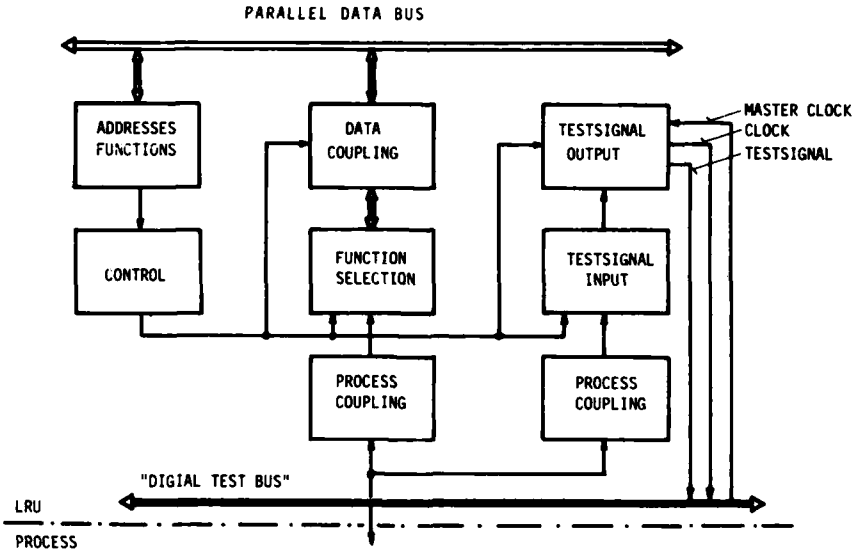
DORNIER FIG 2: TESTLEVELS FOR AVIONIC SYSTEMS



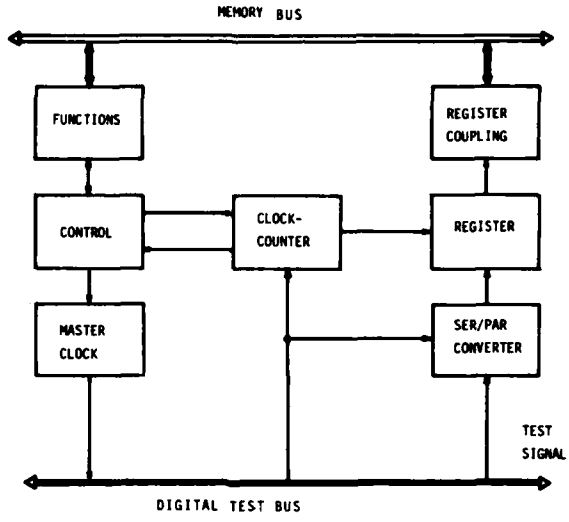
DORNIER FIG.3: GENERAL CONFIGURATION OF A LRU MUDAS-SUBSYSTEM



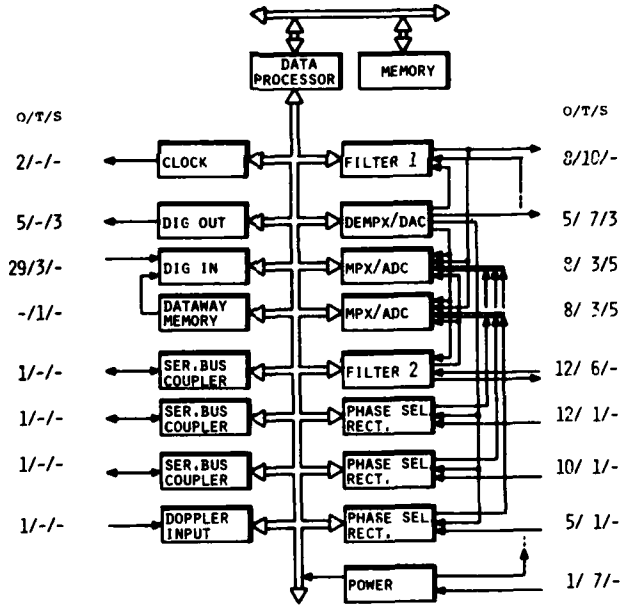
DORNIER FIG. 4: STRUCTURE OF A TESTABLE ANALOG PERIPHERY



DORNIER FIG. 5: STRUCTURE OF A TESTABLE DIGITAL I/O-MODULE

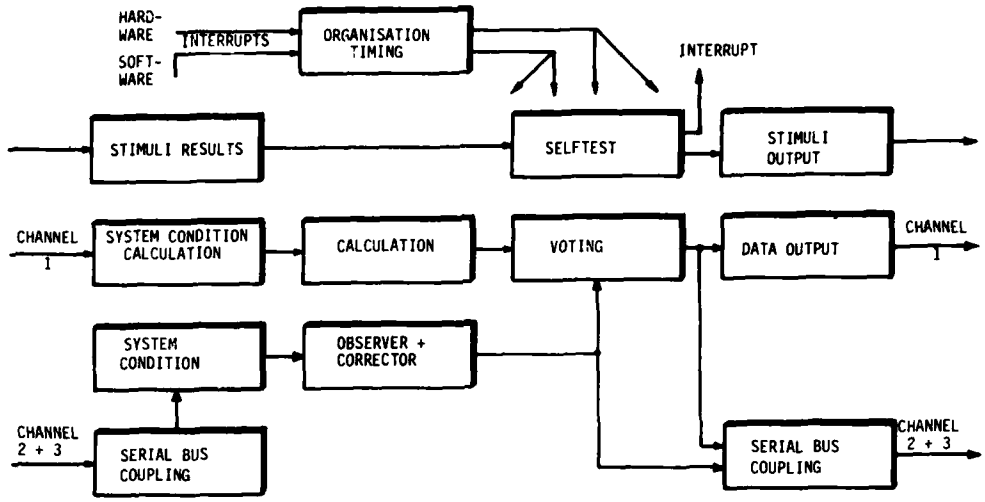


DORNIER FIG. 6: BLOCKDIAGRAM OF A TESTSIGNAL RECEIVER MODULE

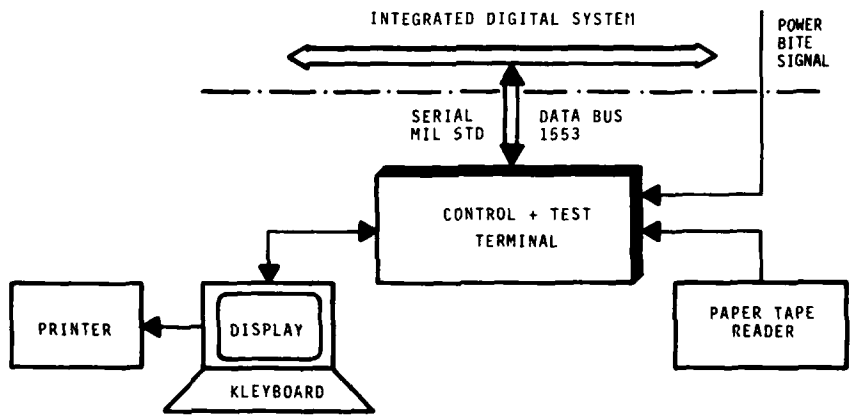


INPUT AND OUTPUT SIGNALS: OPERATIONAL/TEST/SPARES (O/T/S)

DORNIER FIG. 7: LRU OF A FLIGHT GUIDANCE AND CONTROL SYSTEM WITH TESTABLE PERIPHERY



DORNIER FIG.8: STRUCTURE OF THE SOFTWARE OF A LRU



DORNIER FIG. 9: EXTERNAL TEST LEVEL - 2 - TESTEQUIPMENT

Ada

THE UNITED STATES DEPARTMENT OF DEFENSE

COMMON HIGH ORDER LANGUAGE

Dr. David A. Fisher
Office of the Under Secretary of Defense
for Research and Engineering
(Research and Advanced Technology)
3D1079 Pentagon, Washington DC 20301

The United States Department of Defense (DoD) spends more than three billion dollars a year on computer software. This includes the design, development, acquisition, management, and operational support and maintenance of such software. Only a small fraction of this effort is involved with the accounting, inventory, payrolling, and financial management functions which are defined by the Federal Government as Automatic Data Processing, those functions that have their exact analogy in the commercial sector and share a common technology, both hardware and software. A much larger fraction of the DoD's computer investment is in computer resources which are embedded in, and procured as part of, major weapons systems, communications systems, command and control systems, etc. In this environment the DoD finds itself spending an even larger share of its systems resources on software. As a result, this area is receiving increasing attention from the highest levels of management. A number of technical and managerial initiatives have been called out to both reduce the cost and improve the quality of Defense systems software. A management plan has been formulated in this area and initial guidance is provided by DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems.

In the area of software we may have, at the present time, more flexibility and a greater influence on the technology than with hardware. Some years ago, the DoD was a major innovator and consumer of the most sophisticated computer hardware. It now represents only a small fraction of the total commercial market. In software, that unique position still maintains. A significant fraction of the total software industry is devoted to DoD related programs and that is true in even larger proportion for the more advanced and demanding systems. Thus, there is both an opportunity and a responsibility in the software arena which is past for hardware.

One specific initiative which has been called out by DoD Directive 5000.29 is the use of high order languages (HOL) in systems development. The advantages are well known and in many communities, for instance, the COBOL financial management community or the FORTRAN scientific computational community, these advantages are so persuasive that there has been essentially no alternative to the use of these common languages for more than a decade. The obvious advantages include ease of writing of programs, self-documentation, ease of maintenance, ease of modification, transportability of programs, simplification of training, etc.

It is surprising that a general consensus has not mandated a common high order language for embedded systems. There are, however, a number of managerial and technical constraints that have acted against this in the past. For most Defense systems applications, very severe timing and memory considerations have been prominent in the past, often governed by real time interaction with the exterior environment. Because of these constraints, and restrictions in developmental cost and time scale, many systems have opted for assembly language programming. This decision is often substantially influenced by past experience with poor quality compilers and the fact that the assembler comes with the machine, while the compiler and its tools usually must be developed after the project has begun. The advantages of high order languages, however, are compelling and many more recent systems developments have turned to HOLs. Because of limitations of available high order languages, the programs generated most often include very large portions done in assembly code and linked to an HOL structure, negating many of the expected advantages.

Further, many systems have found it convenient to produce their own high order language or some perhaps incompatible dialect of an existing one. Since there is no general facility for control of existing languages, each systems office has had to do the configuration control on their language and compilers and continue to maintain such on their particular dialect through the entire maintenance phase of the system, which may be very long. This has reduced the contractual flexibility of the government and restricting competition in maintenance and further development. This lack of commonality negates many advantages of high order languages including transportability, sharing of tools, the development of very powerful tools of high efficiency and, in fact, not only raises the total cost of existing tools, but in some cases essentially prices them out of the market. Many development projects are very poorly supported and forced to live with a technology which is far below the state-of-the-art.

By the early 1970's each of the military departments had underway studies or actual language designs which were expected to lead to common languages for large portions of those departments, in January 1975 the Director of Defense Research and Engineering set up a DoD-wide program with the goal of a single common military computer programming language for embedded systems. The intent was to have a real time language to supersede

those numerous ones in current use. Further, to assure non-profliferation during the duration of this effort, all other DoD sponsored implementations of new high order languages were halted. In January 1975, a High Order Language Working Group (HOLWG) with representation from the Military Services and DoD Agencies was established as the agent for this effort.

Briefly, the logic of this initiative is as follows:

- o The use of a high order language reduces programming costs, increases the readability of programs, eases program modification, facilitates maintenance, etc. and generally addresses many of the problems of life cycle program costs.

- o A modern high order language performs these tasks better and, in addition, features which improve readability and understandability can be included in the language designed in such a way that they also can be used to aid automatic test generation, analysis and verification as well. A modern language is required if real time, parallel processing, input/output and error recovery portions of the program are to be expressed in high order language rather than in assembly language inserts which destroy most of the readability and transportability advantages of using an HOL. A modern language may also provide better error checking, more reliable programs, and the capability for more efficient compilers.

- o Many of the advantages of a high order language can only be realized through accompanying software tools. A total programming environment for the language includes not just compilers and debugging aids but text editors, interactive programming assistance, automatic testing facilities, automated program analysis tools, incremental modification tools, extensive module libraries, and a variety of compiler options including code optimization. Wide use of such tools, which are often unavailable today, through use of a common language would significantly reduce the life cycle cost of software. Development of newer more powerful tools holds even greater promise. Unfortunately, the average programmer's tool box is rather bare. Because of the time, difficulty, and high cost involved in preparing these tools for each new combination of language, machine, and operating system, only the very largest projects have been able to assemble even a representative set. Smaller projects must be content to develop the same set of primitive tools over and over. While in many cases development of tools can be shown to be desirable in the long run, day to day pressures usually prevail. There is almost never time to do it right. The use of a common machine independent high order language across many projects and controlled at some central facility, would allow sharing of resources in order to make available the more useful, more powerful, and more expensive tools which no single project could generate or support. At the same time, it would make those previously generated tools available at the beginning of a project, reducing both start up time and risk.

- o Reducing the number of languages supported to a minimal number, therefore, provides the greatest economic benefit. There are, of course, costs associated with supporting any particular project and general costs of supporting any language. For a sufficiently large number of users, presumably the costs would be proportionally less. Perhaps 200 active projects contributing to a single support facility may not be proportionally much cheaper than two facilities each supporting 100 projects, although the absolute saving would be significant.

- o There are, however, unique advantages to having a singly military computer language. With a single language, one could reasonably expect new computers proposed for a project to be supplied by the manufacturer with a compiler. This is, in fact, the experience of the British with their common language effort. If there were five or ten common languages, such is not a reasonable expectation. In fact, if there were a single common language, its use in DoD and the provision of tools by the DoD would make it a popular candidate for use elsewhere. Sufficient use could be generated that it would be economically sound to produce machines with firmware targeted to this high order language, thus further decreasing cost and increasing efficiency. The multitude of military languages in the past has not received this sort of acceptance. A single modern well supported machine independent high order language might even be expected to influence academic curricula, improving the training not so much of individual programmers but the understanding and capabilities of the general engineering community for support of DoD programs.

The High Order Language Working Group (HOLWG) was chartered to formulate the requirements for common DoD high order languages, compare those requirements with existing languages, and recommend adoption or implementation of the necessary common languages. For the very near term, administrative recourse was taken. DoD Directive 5000.29 specifies that "DoD approved high order programming languages will be used to develop Defense systems software unless it is demonstrated that none of the approved HOLs are cost effective or technically practical over the system life cycle... Each DoD approved HOL will be assigned to a designated control agent..." Thus, the use of high order languages is established and indeed very strongly mandated, since life cycle costs are usually dominated by maintenance where the high order languages have considerable advantage over assembly language. Approved high order languages will be used, thereby reducing the proliferation and further, these

languages will be controlled by central facilities. DoD Instruction 5000.31, Interim List of DoD Approved High Order Programming Languages, designates CMS-2, Jovial J73, FORTRAN, COBOL, Jovial J3, TACPOL, and SPL/1 as the only currently approved languages and assigns control responsibility.

Formalization of the approved languages was a major step forward and recognized for the first time the corporate commitment of the Department of Defense to provide long term support for languages. It stops the proliferation of languages in that all new systems are to be programmed in one of these languages, but there is no intent that already existing programs be redone or that the projects, already committed to a language, change. There are, however, limitations. The languages themselves are selected from the present Service inventories and are not, in general, modern powerful languages. They are generally deficient in tools and in availability of compilers. They are seldom machine or operating system independent. Further, we have only started on the concept of control. It will be some time before they reach the state of a rigorously defined, well supported and controlled language. They are, therefore, a very near term interim solution. A more satisfactory technical solution to the problem is to formulate requirements, evaluate the existing languages, select the best for modification to meet the requirements, and build a single common high order language, if that proves technically feasible.

The first charge to the High Order Language Working Group was to establish requirements. This working group was to consider general purpose computer programming languages, those which are used by a programmer to specify computations to a computer, that is, one of the level of the interim approved languages. This is a limited goal which does not include conversational application packages nor special purpose languages such as requirements specification languages, query languages, job control languages, automatic test equipment languages, or simulation languages, that do not provide a general purpose computing capability.

The goals of such a high order language are well agreed upon.

- One wishes to have the language facilitate the reduction of the cost of software. This cost must be reckoned on the total burden of the life cycle, including maintenance and certainly not just the cost of production or program writing.

- Transportability allows the reusing of major portions of software and tools from previous projects and the flexibility to modify hardware configurations.

- The maintenance of very long lived software in an ever changing threat situation requires responsiveness and timely flexibility.

- Reliability is an extremely severe requirement in many Defense systems and is often reflected in the high cost of extensive testing and verification procedures.

- The readability of programs produced for such long term systems use is clearly more important than coding speed.

- The general acceptability of high order languages is determined, at this time, by the efficiency and quality of the compiled code. While rapidly falling costs of hardware may make this difficult to substantiate in general, each project manager will compare the efficiency of the object code produced against an absolute standard of the best possible machine language programming. Very little degradation is acceptable.

While these and similar goals are well accepted, they do not lend themselves to a quantifiable or rational assessment of languages. Alternatively, one could establish criteria which were excessively explicit, determining the form but not necessarily the capability of the language. Rigorous defintion of the exact level of requirement proved difficult. Therefore, a STRAWMAN of preliminary requirements was established to define this level by illustration. The STRAWMAN was forwarded to the Military Departments, other government agencies, the academic community and to industry. Additionally, a number of technical experts outside the U.S. were solicited for comments, the European and NATO community being especially responsive.

The review of the STRAWMAN resulted in inputs from which were put together a fairly complete, but still tentative, set of requirements called the WOODENMAN. This too was widely distributed for comment. Based on various inputs and the official responses from each of the Military Departments, a TINMAN set was derived which then represented the desired characteristics for a high order computer programming language for the DoD.

Early in this program, there was the feeling that different user communities might have fundamentally different requirements with insufficient overlap to justify a common language between them. Such communities include avionics, weapons guidance, command and control, communications, tactical systems, and training simulators. The surprising result was that the technical requirements so generated were identical. It was impossible to single out different sets of requirements for different communities. All users needed input/output, real time capability, strong data typing for compiler checking, modularity, etc. Upon reflection, the technical rationale for this was clear. The surprise was historical, based on the observation that in the past the different communities had favored different language approaches. Further investigation showed that the origin of this disparity was primarily administrative rather than technical, and the result that a single set of requirements would satisfy a broad set of users became less of a surprise. This did not, however,

establish that a single language could meet all the stated requirements, only that, if a language meeting all the requirements existed, it would satisfy the users needs.

Very wide distribution of the TINMAN followed and for a year comments were received on this document. An international workshop was held at Cornell University in the fall of 1976 to illuminate the current state of the art of programming language design and implementation. In January, 1977, a new version called the IRONMAN was issued. It was essentially the same set of requirements as the TINMAN, modified slightly for feasibility and clarity, but presented in a different format that simplified analyses for technical feasibility. The TINMAN was discursive and organized around general areas of discussion. The IRONMAN, on the other hand, is very brief and organized like a language description or manual. It provides a specification with which to initiate the design of a language. It remains sufficiently general to avoid specifying particular features or structures, while still giving the needed capabilities. The IRONMAN was revised in July 1977 and again in June 1978 to form the final version which is called STEELMAN. These revisions were mainly to clarify the intent, but also corrected a few errors and inconsistencies that were identified lately.

The next phase of the work was the evaluation of existing laguages. This was begun in a formal fashion in the summer of 1976, at which time the current requirements document was the TINMAN. Differences between the TINMAN and the IRONMAN are sufficiently minor so as not to affect the conclusions of this evaluation. The purposes of the evaluation were: to examine the existing languages and determine if one or a combination could satisfy the requirements; to determine on the basis of evaluation of existing languages whether the requirements themselves were feasible and valid; to determine if it was within the state-of-the-art to have a single language satisfying all these requirements; and to recommend the procedure for arriving at the desired minimal set of languages. The languages included in the evaluation were those nominated to the Interim Standard List, languages in wide acceptance elsewhere, and certain modern languages offering advanced capabilities. The main set of languages was evaluated very formally through contracts in which each language was evaluated by more than one contractor and each contractor had several languages to evaluate, thus giving a cross check on the results. In addition, a number of individuals submitted detailed evaluations of specific languages with which they had a unique familiarity. All these evaluations consisted of a comparison of the language against each individual point of the TINMAN. They were not mere existence checks but the languages were also examined for feasibility of modification should a particular point not be met and for possible deletion of features not needed to satisiy the TINMAN requirements. The following languages received formal evaluations: FORTRAN, COBOL, PL/1, HAL/S, TACPOL, CMS-2, CS-4, SPL/1, J3B, J73, ALGOL 60, ALGOL 68, CORAL 66, PASCAL, SIMULA 67, LIS, LTR, RTL/2, EUCLID, PDL2,PEARL, MORAL, EL-1. Besides those languages receiving formal evaluation, a number of other languages were examined for specific features or as examples of modifications of these languages and contributed data on the feasibility and flexibility of the various language approaches.

Such was the bulk of these studies that a government committee was put together to analyze and compare the evaluations and to make recommendations consistent with them. These conclusions and recommendations were adopted unanimously by the High Order Language Working Group as the basis for the next phase of the project. The conclusions may be briefly summarized as follows:

- Among all the languages considered, none was found that satisfies the requirements so well that it could be adopted as the common languages.
- All evaluators felt that the development of a single language satisfying the requirements was a desirable goal.
- The consensus of the evaluators was that it would be possible to produce a language within the current state-of-the-art meeting essentially all the requirements.
- Almost all the evaluators felt that the process of designing a language to satisfy all the requirements should start from some carefully chosen base language.
- Without exception, none of the interim approved languages was found by the evaluators to be appropriate to serve as a base for the development of a common language for embedded military applications.
- Several languages were found to be appropriate as a base for modification. All such languages were derivatives of one of three languages: PASCAL, ALGOL-68, or FORTRAN.

At this point we had determined, as well as can be done on the basis of paper studies without actual construction of a language, that a single language could be constructed to meet the requirements, further, that this could be done with elements which are mutually consistent and within the demonstrated state-of-the-art. The next step in the project was, therefore, to provide a preliminary definition of a language. Alternatively this might be considered an elaborate feasibility proof. Such definition was to be informal but fairly complete and to consider the cost and nature of implementations.

The preliminary definition used the Revised IRONMAN as the requirements specification and drew upon the previous work. Multiple competitive contracts were used with the best

products to be selected for continuation to full rigorous definition and developmental implementation. Each design was to be produced by a small closely knit team under the control of one person.

In August 1977, four contracts were awarded to produce competitive prototypes of the common high order language. These awards came as a result of a request for proposal and offers received from fifteen firms, both U.S. and foreign. The successful contractors were CII-Honeywell Bull, Intermetrics, SofTech, and SRI-International.

While different approaches were offered, all four winning contractors proposed to use PASCAL as a base, thereby restricting the products in form and making it somewhat easier to compare the results. We were prepared to deal with three different base languages, so the outcome was coincidental. It should be noted however that the requirements against which the language was designed were not the same as those driving PASCAL. Thus, only a family resemblance between PASCAL and the design product could be expected.

The products of Phase I, the preliminary designs, were received in February 1978. The considerable interest that this project has generated in the outside community made it possible to seek technical input for the evaluation of these designs from the industrial and academic communities worldwide. Eighty volunteer analysis teams were formed and produced extensive technical analysis of the designs. The period available was quite short, but the designs were only preliminary and the purpose of analyses was to determine which should be continued to completion. On the basis of these analyses, CII-Honeywell Bull, and Intermetrics were selected to continue and resumed work in April 1978. As a result of both the designs and the analyses, the requirements were updated in June 1976 to the STEELMAN version. Since this may logically be the final set of requirements, some care was taken to remove apparent misunderstandings and discrepancies which surfaced as the result of the actual design of the four languages. The exceptionally rigorous review of the languages by the analysis teams in the context of the requirements was a further test.

The second phase of the design produced a complete language manual, a design rational document and a limited test ranslator. The test translator was intended only as an aid in testing the design of the language and was neither complete nor production quality. Based on additional public review and analysis and a workshop with joint discussions among the design teams, the analysis teams and DoD participants, a final selection was made in May 1979. The Green language designed by CII-Honeywell Bull was chosen for further rigorous testing and continued refinement, and thus became the initial design of Ada.

At the same time, DoD sponsored three different economic analyses of the common language effort. These were targeted to questions of expectation of savings to result from the successful completion of the program. They further examined various introduction strategies and rates of introduction of Ada. Not only were significant savings identified, but they were shown to be magnified as a function of the rapidity with which Ada could be introduced. These analyses were not however based on the technical merits of the language or on its suitability to military applications, but only on its machine independence and wide availability.

To verify that the Ada language design would adequately support the range of embedded computer applicatons which motivated "STEELMAN," a one year test and evaluation was initiated in May 1979. An open invitation was issued for volunteers to choose an existing application, preferably written in some high order language, and implement it in Ada. Each of the services identified teams of programmers to implement applications considered critical to their own embedded systems. Teams from industry, academia and government volunteered to conduct independent appraisals.

An Ada orientation course was offered to Test and Evaluation participants in the early summer. The one week session, led by the language design team, was presented at the U.S. Air Force Academy, the Naval Postgraduate School, the U.S. Military Academy, the Georgia Institute of Technology, and the Shivenham Royal Military College. Participants were introduced to both the philosophy of design and to specific language features.

A test and Evaluation Workshop, jointly sponsored by DARPA and MIT was held in Boston on October 23-36, 1979. Some one hundred participants gathered to discuss issues ranging from simple transliteration (from some high order language to Ada) to significant refinement activities. The mix of applications varied from straight forward data processing to complex control of real time systems involving synchronization of parallel processes. Although a number of specific language issues were raised, it became apparent that Ada is adequate for all applications attempted. The general theme expressed by a number of speakers was that, while the language is both adequate for their applications and a significant improvement over existing embedded computer languages, there are some important refinements that are needed.

Although no special environment is needed to use Ada, it was realized early in the development process that acceptance of the language and ultimate payoff would be magnified by the development of a useful and powerful support environment. A workshop, jointly sponsored by the Army, Navy, Air Force and the University of California - Irvine, was conducted at the Irvine campus June 20-22, 1978 to initiate discussion of alternatives for environments.

From this workshop an initial environment specification, called "Pebbleman," was developed. Pebbleman described all aspects of the Ada language environment (ALE) including language

standards, policy, configuration control, compiler validation, software tools and management tools. Pebbleman was widely distributed in July 1978, and revised in January 1979.

At this stage, the HOLWG decided to separate the technical issues from the policy issues. After several informal iterations and reviews a set of technical requirements for an Ada Program Support Environment (APSE) was distributed in November 1979 as the "Preliminary Stoneman".

To better understand and define these requirements, the HOLWG sponsored an Ada Environment Workshop, November 27-29, in San Diego. Two hundred twenty industrial, research, and government participants discussed relevant features of existing environments from both the users and developers points of view. From these discussions and written responses to the "Preliminary Stoneman" the "Stoneman" document is being prepared and will be distributed this winter. Stoneman is a requirements document which specifies the structure and content of an APSE to support both the development and maintenance phases of a system, requirements are stated for the support system on a host machine and the run-time considerations for the target machine.

The APSE is to offer a well-coordinated set of tools with uniform interfaces to support a programming project throughout its life cycle activity. It must be highly portable and employ uniform conventions for interface between user and tool. Stoneman introduces the notion of a common open-ended database to serve as the interface through which a highly modular set of software tools can communicate. This database will maintain information important to such functions as version control, library support and project management. The form and content of the database are not specified but the Stoneman calls for the selection of a set of conventions.

It is likely that more than one APSE will evolve. Therefore a Kernel Ada Programming Support Environment (KAPSE) is defined to provide a virtual support environment. The KAPSE is the environment made available to the APSE tools to ensure a machine-independent interface. All APSE tools using a common KAPSE should prove portable over the set of environments supported by that KAPSE.

Stoneman also defines a minimum set of functions which an APSE should perform. This minimal APSE (MAPSE) must provide a method to create database objects, modify database objects, produce new objects which are records of analysis of other objects, transform an object from one representation to another, support the display of objects, parse, link, load and execute.

The Air Force has issued a draft RFP specifying the competitive design of an APSE. The "Stoneman" is a supporting document in the RFP and the HOLWG will continue to work with the Air Force in better defining an APSE.

Introduction of Ada should involve more than just learning a new language. It offers an opportunity to provide training in modern programming methods that are appropriate to Ada, but inappropriate to lower level languages, and often unfamiliar to DoD programmers (whether in house or contractors). Introduction of the language will permit the use of new concepts and facilities, some, for which there is little current experience.

Realizing that Ada presents a novel opportunity to present a coordinated view of modern programming practice, complete with a language and support environment, the HOLWG established an Advisory Committee on Education and Training in March 1979. The commmittee, composed of military and civilian educators, is to coordinate education and training activities to ensure an orderly and coordinated introduction of Ada.

The goal is to develop a base of experience from which to launch the introduction of Ada within DoD. Members of the committee are actively engaged in teaching and coordinating Ada courses in universities and industry. Based on the experience gained from these efforts, and committee intends to develop a model course from which other courses may be derived. It is clearly appropriate for courses to be oriented to the experience of the student.

Courses with titles such as "Ada for FORTRAN programmers," "Ada for Pascal Programmers," Ada for Machine Language Programmers," will undoubtedly appear. These courses will focus on the similarities and differences between Ada and some commonly understood language, some of which share little of the philosophy of the Ada design. Such courses will not naturally examine the motivation for features nor promote application of the relevant principles. The model course should help infuse appropriate concepts into such specialized courses.

Courses dealing with Ada related issues were offered during the fall term at Carnegie-Mellon University, New York University, Stephens' Institute of institutions, both university and industry plan courses in the coming terms. Based on the collected experience from these efforts, a design for a model course will be developed and distributed for comment. The goal is to provide a model course which provides coordinated treatment of the languages, complete with instructional materials, by early summer. It is hoped that this model will be of assistance to anyone preparing an Ada course.

The common language effort has not attempted to solve the software problem, but rather to provide a leverage for emerging solutions and to eliminate certain nspicuous and unnecessarily duplicative costs. Somewhat surprisingly, it has been possible to satisfy

substantially all the identified requirements without encountering any significant technical difficulties. This may be the result of setting our sights on what we know. George Washington didn't ask for airplanes or atomic bombs or lasers, all he wanted was muskets, cannon and sabers. Future language research is vital if we are to be able to deal with ever more complex and demanding military systems, if we are to be able to satisfy the increasingly severe systems reliability requirements, and if we are to significantly impact the high cost of software maintenance. It is not the intent that the existence of this language stifle such research, rather that it provide a target and a user, a data and requirements gathering agent, and clearer identification of underlying software problems in military and real time applications.

Besides the normal interaction between portions of the Department of Defense and other agencies of the U.S. Government, this effort has had close relations with and received a great deal of support and technical input from a number of outside organizations with similar aims. The appropriate subcommittees of the American National Standards Institute and the International Standards Organization including their Working Group on Programming Languages for the Control of Industrial Processes have been kept closely informed of this work. The International Purdue Workshops on Industrial Computer Systems have long held an interest in this area and in particular an affiliate group, Long Term Procedural Language-Europe (LTPL-E) has as a goal the production of a language much like the one we desire. The goals of this gorup have recently been adopted by the European Economic Community and there has been a very intimate relationship between this group and the HOLWG. This is perhaps the most closely analogous group, trying to satisfy the requirements of several countries in several real time applications areas. Perhaps the most successful national common language effort has been that of the British Ministry of Defense in specifying language CORAL 66 for all MOD real time applications. The HOLWG has received much valuable technical and managerial insight from the British experience and to enhance this cooperation, the British assigned a senior technical expert to the HOLWG to be resident in Washington, providing both technical input and liaison. More recently, both the German and French governments have initiated procedures to standarize on existing high order languages, PEARL and LTR, respectively. The Federal Republic of Germany also assigned a technical representative to the HOLWG in Washington. The Japanese government, Ministry of Information, Technology and Industry, is subsidizing a consortium to produce a software production environment, central to which is a common programming language. The CCITT has proposed a common high order language for international use in communications.

It appears that the time is ripe for moving to a common High Order Language both technically and administratively, but significant milestones do remain.

Several efforts are just now getting underway for the introduction of Ada. Delivery of the final language design is expected in May with formal standards established in June 1980. In anticipation of the development of production compilers. DoD has undertaken a contract with SofTech to develop an Ada Compiler Validation Capability (ACVC). This capability will aid compiler builders to ensure that their products satisfy the Ada standards, and will provide an extensive set of test programs to determine compatibility with the standard. The initial ACVC will be delivered in June 1980 and will be used by chartered Ada Compiler Validation Facilities (ACVF) to test and certify compilers.

The first production compiler contract from the DoD will be let by the Army in February, 1980. It calls for the development of an Ada compiler written in Ada, having separable front end and code generators, and for multiple code generators. This will be a two year effort. The Air Force contract to design and implement an APSE will also include a production compiler. This compiler will also be written in Ada, have separable front end and code generators, and have an intermediate representation compatible with that of the Army compiler. The code generators will however be targeted to several machines widely used in the Air Force. In addition there are several research compilers which emphasize specific goals such as target code optimization or clarity of presentation without commitment to a production quality product. In all cases the compiler must pass the testing and certification process prior to use on DoD projects.

Recently the British Government, the German MOD, and the European CEC have made certain commitments to the introduction of Ada. The DoD will continue to cooperate fully with other Ada user organizations to insure that all compilers are fully compatible with a single standard definition, that the advantages of standarization are not lost through proliferation of dialects, and that they are magnified through the broadest possible sharing of resources. The High Order Language Working Group actively solicits comments and cooperation in maximizing the success of this effort.

BIBLIOGRAPHY

Documents with AD numbers are available from the National Technical Information Service. Most other current DoD publications can be obtained from DARPA.

"Proceedings of the Ada Environment Workshop," Harbor Island, San Diego, November 27-29, 1979.

"Department of Defense Requirements for Ada Lanugage Integrated Computer Environments -- Preliminary STONEMAN," DoD, November 1979.

"Initial Thoughts on the Pebbleman Process," IDA Paper P-1392, David A. Fisher and Thomas A. Standish, June 1979.

"Preliminary Ada Reference Manual," SIGPlan Notices, Vol 14, No 6, June 1979.

"Some Observations Concerning Existing Environments," Peter F. Elzer, Dornier Systems GmbH, May 1979.

"Proceedings of the Ada Test and Evaluation Workshop," Museum of Science, Boston, October 23-26, 1979.

"Department of Defense Requirements for the Programming Environment for the Common High Order Language -- Revised PEBBLEMAN," DoD, January 1979.

"The U.S. Department of Defense Common High Order Language Effort," Lt.Col. William A. Whitaker, DARPA, September 1978.

"Department of Defense Requirements for the Programming Environment for the Common High Order Language -- PEBBLEMAN," DoD, July 1978.

"Interim Ada Configuration Management Plan," HOLWG, July, 1978.

"Proceedings of the Irvine Workshop on Alternatives for the Environment, Certification and Control of the DoD common High Order Language," University of California, Irvine, June 20-22, 1978.

"Department of Defense Requirements for High Order Computer Programming Languages -- STEELMAN," DoD, June 1978.

"DoD High Order Language Commonality Effort - Phase I Design Report," HOLWG, June 1978, ADB-950587.

"Report of the Eglin Workshop on Common Compiler Technology," ADTC.

"Studies of the Economic Implications of Alternatives in the DoD High Order Commonality Effort," HOLWG.

"Benefit Model for High Order Languages," TR78-2-72, Joseph M. Fox, Decisions and Designs Incorporated, March 1978.

"DoD's Common Programming Language Effort," David A. Fisher, IEEE Computer, Vol 11, No 3, March 1978, pp 24-33.

"Rational for Fixed Point and Floating Point Computational Requirements for A Common Programming Languages," P-1305, David A. Fisher and Philip R. Wetherall, Institute for Defense Analyses, January 1978.

"Plan for the Analyses of the Preliminary Designs for A Common Programming Language for the Department of Defense," Defense Advanced Research Projects Agency, December 30, 1977.

"Design and Implementation of Programming Languages - Proceedings of a DoD Sponsored Workshop," October 1976, John H. Williams and David A. Fisher, Eds., Lecture Notes in Computer Science, Vol 54, 496pp, Springer-Verlag, 1977.

"The Common Programming Language Effort of the Department of Defense," D.A. Fisher, Computers in Aerospace Conference, November 1, 1977.

"A Cost/Benefit Analysis of High Order Language Standardization, M78-206, J.A. Clapp, E. Loebenstein and P. Rhymer, The MITRE Corporation, September 1977.

"Defense System Software Research and Development Technical Plan," DoD, September 1977.

"Department of Defense Requirements for High Order Computer Programming Languages - Revised IRONMAN," HOLWG, July 1977.

"Language Evaluation Coordinating Committee Report to the High Order Language Working Group," S. Amoroso, P. Wegner, D. Morris, and D. White, 2617pp, HOLWG, January 14, 1977, AD-A037634.

"Department of Defense Requirements for High Order Computer Programming Languages - IRONMAN," HOLWG, January 1977.

"Interim List of DoD High Order Programming Languages," DoD Directive 5000.31, November 24, 1976.

"A Common Programming Language for the Department of Defense - Backgroun and Technical Requirements," D.A. Fisher, Institute for Defense Analyses, P-1191, June 1976, AD-A028297.

"Department of Defense Requirements for High Order Computer Programming Languages - TINMAN," HOLWG, June 1976.

"Charter for the High Order Language Working Group," Management Steering Committee for Embedded Computer Resources (MSC-ECR).

"Management of Computer Resources in Major Defense Systems," Department of Defense

Defense Directive 5000.29, April 26, 1976.

"Defense System Software Mangement Plan," March 1976, AD-A022558.

"An Introspective Analysis of DoD Weapon System Software Management," Barry C. DeRoze, Defense Management Journal, Vol 11, No 4, pp. 2-7, October 1975.

"DoD High Order Programming Language," Memorandum by Director, Defense Research and Engineering, January 28, 1975.

"Embedded Computers - Software Cost Considerations," John H. Manley, AFIPS 1974 National Computer Conference (NCC) Proceedings, Vol 41, pp. 343-347.

"Automatic Data Processing Costs in the Defense Department of Defense," P-1046, Institute for Defense Analyses, October 1974, AD-A004841.

"Technology Trends: Software," Information Processing/Data Automation Implication of Air Force Command and Control Requirements in the 1980s (CCIP-85), Vol IV, Space and Missile Systems Organization, AFSC, October 1973.

"Highlights, Information Processing/Data Automation Implication of Air Force Command and Control Requirements in the 1980s (CCIP-85)," Vol I, Revised edition, Barry W. Boehm et. al., Space and Missile Systems Organization, AFSC, February 1972.

NOTE: Significant portions of this paper were contributed by Lt.Col. William A. Whitaker, USAF, and Lt. Col. Larry Druffel, DARPA.

COMPILER WRITING TECHNIQUES FOR AVIONICS APPLICATIONS

by
Raymond J. Rubey
Barry L. Wolman
SOFTECH, INC.
460 Totten Pond Road
Waltham, Massachusetts 02154
U.S.A.

SUMMARY

This paper reviews some of the options in compiler construction for avionics applications and describes the structure of a typical compiler. Compilers for avionics applications have many similarities and a few significant differences as compared with compilers used in general-purpose applications. This paper will concentrate on the differences.

INTRODUCTION

The use of Higher Order Languages (HOLs) for developing avionics software is becoming the usual practice. Many HOLs have been defined with avionics applications in mind; these include JOVIAL J3B[1], JOVIAL J73[2], SPL/I[3], CMS-2[4], HAL/S[5], CORAL-66[6], PEARL[7], and Ada[8]. Regardless of the HOL used, a compiler is needed to translate the program written in the HOL (i.e., the source code) to the machine or assembly language (i.e., the object code) of the avionics computer. These compilers are complex computer programs; the availability and characteristics of the compiler for the selected HOL have a major influence on an avionics software development effort. Because compilers are expensive and require considerable time to develop, many avionics projects are inhibited from using a HOL. There may be neither the funds nor the time available to the avionics project to develop a compiler. Indeed, the availability of a proven, efficient compiler may be the prime consideration in the selection of the particular HOL to be used in an avionics project.

HOST AND TARGET COMPUTER DEPENDENCIES

Because a compiler is a computer program, it must execute on a specific digital computer called the host computer. The compiler translates the higher order language into the machine or assembly language code of a specific computer called the target computer. Most general-purpose compilers execute on and generate code for the same computer; that is, the host and target computers are the same. For example, the IBM 370 FORTRAN compiler executes on and generates code for the IBM 370 computer. In avionics applications, the host computer is usually a large- or medium-scale general purpose computer while the target computer is a different, smaller, airborne or embedded computer. For example, the JOVIAL J3B compiler for the F-16 Fire Control system is hosted on the IBM 370 and generates DELCO M362F code. This type of compiler is called a cross-compiler. Cross-compilers are used in avionics applications because the capacity, the peripheral equipment, and support software of the avionics computer are not adequate for a compiler. In addition, the capabilities of larger general-purpose computers facilitate compiler use, enable the creation of a more efficient compiler, and permit the effective use of support tools such as a formatter or a cross referencer. In particular, a cross-compiler can employ more elaborate optimization techniques than a compiler hosted on a small computer of limited capabilities.

The use of a HOL for an avionics software project means that there must be a cross-compiler targeted to the selected avionics computer hosted on a computer available to the software development team. If such a compiler does not exist, there are several ways that an appropriate cross-compiler can be obtained. (We rule out the possibility of selecting a different target or host computer since these decisions are difficult to change.)

If a compiler for the HOL is available on the host that generates code for a different target computer, that cross-compiler can be converted to generate code for the desired target by a process known as retargeting. This is a straightforward process and it is common for a set of cross-compilers for a particular HOL to be implemented on the same host computer. For example, the SofTech developed J3B compiler is hosted on the IBM 370 and generates code for the IBM 370, Delco M362F, Litton 4516, Singer SKC 2070, IBM 4Pi, and IBM AP/101C.

If a cross-compiler for the HOL targeted to the desired avionics machine is not available on the support computer but is available on a different host, that cross-compiler can be converted to run on the desired host by a process known as rehosting. If the cross-compiler is written in assembly language, rehosting can be an expensive operation because the entire cross-compiler must be translated into the assembly language of the new host. If the cross-compiler is written in a HOL, such as the HOL itself, rehosting can be substantially simplified. In this case, rehosting involves compiling the cross-compiler with a version of the compiler that generates code for the new host computer. It is not unusual for a compiler written in its own language to be hosted on many machines. For example, the SofTech developed AED compiler runs on the IBM 370, CDC 6600, and UNIVAC 1110 series and is targeted to many more machines.

Usually the general-purpose computer (e.g., IBM 370, DEC 10, UNIVAC 1110, CDC CYBER 74, etc.) used as the host for the avionics cross-compiler is also used for other activities such as simulation and modeling in the avionics software development process. It is usually very useful if there is a compiler of the selected avionics language targeted to the host computer as well as to the avionics computer. With this capability avionics programs can be compiled and executed on the general-purpose computer. Thus, initial checkout of the avionics software algorithms can be on a more accessible computer and in a more controlled environment.

After this checkout is complete, code can be generated for the avionics computer rather than for the general-purpose computer by using a different version of the compiler. This can greatly reduce, or eliminate, the recoding that is required when initial algorithm development is done in a different language (e.g., FORTRAN) than that used for the avionics software (e.g., JOVIAL J73). Finally, other benefits, such as increased host-computer transportability, can accrue from using the avionics language for the support software as well as the avionics software development.

EFFICIENCY CONSIDERATIONS

The prime factor that has inhibited the use of HOLs for avionics in the past has been the greater memory required by the slower execution time of HOL-derived programs. The impact of this factor has been reduced for two reasons. First avionics computers of greater capacity but lower cost have made it possible to compensate for efficiency problems in the compiler by providing additional hardware. Second, techniques for producing efficient compilers have evolved. Today, compilers can generate programs no more than 20% slower or bigger than programs coded in assembly language by expert programmers. In some cases, such as the FORTRAN compiler for the CDC 6600, the code produced by the compiler is normally better than the assembly language code most experienced programmers are capable of producing.

The performance characteristics of a compiler depend on the environment in which it will be used. In most situations, a compromise is required between the computer time used by the compiler in translating from source to object code and the time used in execution of the object code. Compilers for general-purpose applications must compile source programs with moderate speed and generate object code of moderate efficiency. Conversely, compilers intended for use in educational environments are constructed to compile very fast at the expense of object code efficiency. On the other hand, compilers for avionics applications must generate object code of the greatest possible efficiency; this efficiency can be obtained even at the cost of a considerable increase in the time required for compilation.

Even with a HOL suitable for the avionics application and an efficient compiler, current practice provides for programming a portion of the software in assembly language. A major reason for this regression to assembly code is the belief that the HOL results in unacceptably inefficient code for some functions, particularly the real-time executive. Other reasons often cited include the inability to use special hardware features and concern over the efficiency of procedure linkage conventions.

We believe that with modern HOLs (and optimizing compilers for them), the use of assembly language should be significantly curtailed, if not eliminated. For example, it seems reasonable to write in assembly language the fault interceptor module that receives control after a hardware fault, but this module should be written to save the machine state and then call an appropriate HOL procedure.

This contention is justified because modern compilers can almost always generate code that is as good as, or better than, assembly code written by average programmers. It will always be possible for the experienced, exceptional programmer to write code that is better than that produced by the compiler for a particular section of source code, but experience shows that such carefully tailored code tends to be hard to change or maintain and leads to high life-cycle costs for the avionics project. A common problem with "efficient" assembly code is that the programmer makes a change in one part of a program but does not make a required change in a logically unrelated section of the program, e.g., because a new register is used. The compiler does not make such mistakes. The compiler never has a bad day and can consider the entire program when generating code.

SofTech has found that the use of target-specific, built-in functions provides a convenient means for giving the HOL programmer access to special features of the target machine. These are predefined functions with code expansions that map onto specific instructions. For example, the functions

```
DOT(A, B, N)
```

and

```
POLY(X, C, N)
```

could be defined for a target machine that had hardware instructions for vector multiplication and polynomial evaluation. On such a machine only a single instruction (with suitable setup) would be used to implement these functions. For another computer lacking such features, the compiler could generate

```
A[1]*B[1] + A[2]*B[2] + ... + A[N]*B[N]
```

and

```
C[0] + X*(C[1] + X*(... +X*C[N]) ... )
```

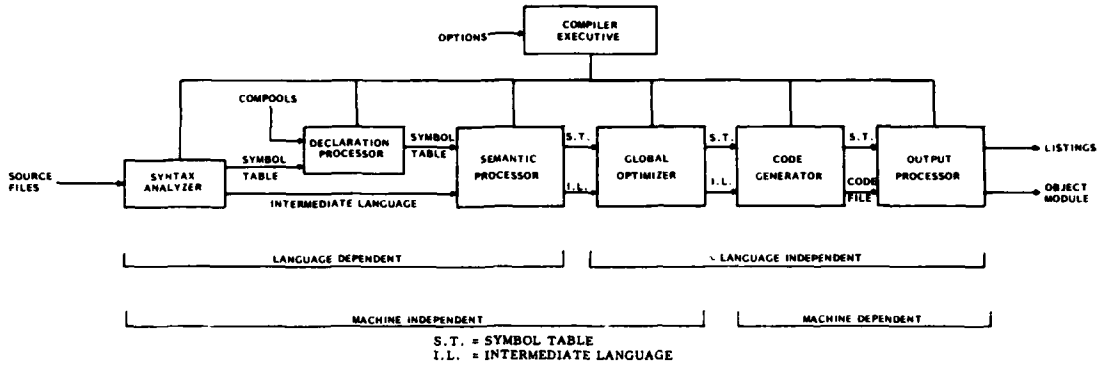
for these two special functions. Such built-in functions enable hardware features to be fully utilized without requiring inline use of assembly code or assembly language subroutines and these functions permit the programs that use them to be moved to other hardware.

Procedure linkage conventions are another area where a good compiler can do as well as, or better than, the assembly language programmer. Some things that can be done in this area are compiling the procedure body as inline code (if space is not at a premium), using a simplified calling sequence, based on information on how the procedure is actually used, and passing formal parameters in registers.

In the few cases where use of assembly language is deemed critical, the assembly code should be provided as a callable procedure that uses the documented calling sequence conventions of the HOL.

COMPILER STRUCTURE

The structure of a typical optimizing compiler for a HOL such as J73 is shown in Figure 1. The compiler consists of a set of sequential phases or modules, each performing part of the translation process. The early phases are normally highly dependent on the particular HOL and almost completely independent (except for parameters such as bits per word) of the target computer. The later phases are highly dependent on the target machine and essentially independent of the HOL being compiled. This partitioning into machine independent and machine dependent sections is the reason development of a cross-compiler is normally a straightforward process. With proper design, retargeting involves the development of only the code generator phases. When compilers that are targeted to both the host computer and an avionics computer are needed, a single set of machine independent phases and two sets of machine dependent phases will provide the needed capability.



The compiler executive module initializes the compiler for operation, processes command lines, opens files, and controls the operation of the remaining modules. Host operating system dependent functions are performed by service routines within the executive. Isolating these functions in the executive makes the rest of the compiler more host independent. Based on user-selected options supplied to the compiler, the executive controls processing by invoking only those modules and functions required to complete a compilation in accordance with the user's selected option. For example, the compiler executive can direct:

- The syntax analyzer module to insert formatting information into the listing file;
- The optimizer to skip optimization processing in order to save compile time for programs whose size is not of concern; this option is also of value during development when the initial code generators are being produced independently of the compiler;
- The optimizer and code generator to skip processing when syntax checking mode is in effect during initial program development.

The compiler executive module also receives error severity codes on completion of each pass that may affect which subsequent modules are called. It skips optimization and code generation when serious errors are detected in the source program during syntax, declaration, or semantic processing.

The syntax analyzer module parses the source program and creates the initial version of the symbol table and intermediate language file. Parsing involves recognition of lexical tokens, such as an identifier, which normally consists of a letter followed by a sequence of letters and digits, and application of syntactic rules to determine the grammatical structure of the source program.

The syntax rules of the HOL are usually expressed in a notation called Backus-Naur-Form (BNF). The BNF description of an assignment statement as a typical HOL would include rules such as

<assign-statement> ::= <reference> = <expression>;

<reference> ::= <identifier> | <identifier> (<ss-list>)

<ss-list> ::= <expression> | <ss-list>, <expression>

These rules state that an <assign-statement> is a <reference> followed by an equal sign followed by an <expression> followed by a semicolon; a <reference> is an <identifier> or an <identifier> followed by a parenthesized list of subscript expressions; and an <ss-list> is one or more <expression>s separated by commas. The BNF description for J73 contains several hundred such rules.

Compilers built by SofTech normally use a table driven parser. The BNF grammar is processed by a support tool that verifies that the grammar is unambiguous (there are no "sentences" with more than one possible parse) and generates a compact set of tables that are used with a bottom-up look-ahead parser. This technique produces syntax analyzers that are as efficient as any that can be hand-programmed; since errors are detected as early as possible, the error recovery properties are good.

The declaration processor module completes the skeleton symbol table created by the syntax analyzer. Information contained in type and variable declarations is checked and appropriate error messages are issued. Machine specific data (localized to a few places in the phase) is used to calculate the size of data and to assign offsets to fields in a record. If the language contains a COMPOOL facility, as does J3B and J73, this phase must also process the COMPOOL files specified by the programmer and extract required declarations.

The semantic processor module performs all semantic checks, introduces type conversions or identifies type incompatibilities, and generates the final intermediate language (IL) representation of the program for use by the optimizer and code generator.

The global optimizer module performs global and target-computer-independent optimizations. These optimizations are performed on the IL representation of the program and a more efficient IL representation is produced. Because of the importance of the optimization operations to an efficient and effective avionics compiler, they are discussed in detail in a later section. The optimization module also inserts information about the use of variables into the IL program to guide the allocation of registers during code generation.

The code generator module processes the IL representation of the source program and maps it into a target machine instruction representation. Instructions are generated from the intermediate language by interpreting each IL operator in terms of target machine instructions. Instructions generated depend on the addressing properties of operands, the availability of registers, and the information on operand usage inserted into the intermediate language by the global optimizer module.

SofTech has successfully used a decision table technique for code generation. Instructions are generated using decision tables, which examine all of these factors using preprogrammed conditions parameterized with the operands and which produce locally optimal code for the appropriate target machine architecture. This technique results in very reliable code generators. A critical optimization activity performed during code generation is the allocation of target computer register stores and reloads. SofTech has used a cost function approach in this register allocation. The cost function uses the information on data usage collected by the optimization module. For each variable, the global significance is a function of the distance to the next reference. A register is selected for each variable reference based on this data. The cost function prioritizes different types of use; intermediate results have the highest priority, followed by reused operands, reused bases, and reused indices.

The output processor module produces all compiler outputs. It performs final instruction optimization and code assembly operations, and prepares the object module in the form of a relocatable binary object file. It prints the source program listing with error messages interspersed and occurring after the source input line that was found in error. It generates a readable description of the complete symbol table and combines the symbol table information for each name with cross-reference information identifying input lines on which each variable is set or referenced. The output processing module produces binary symbol table files for use in COMPOOL input and for potential use by a symbolic debugger. Finally, the output processing module can print statistics about the source program. Some statistics collected by the SofTech J73 compiler are shown in Table 1. These statistics are useful in management review and evaluation, particularly in determining software complexity.

TABLE 1
SOME COMPILER STATISTICS FOR J73

number of symbols;
number of lines;
number of comments;
number of declarations;
number of items of each data type;
number of constants of each data type;
number of procedures and functions at each nesting level;
number of serial and parallel tables;
number of defines;
number of compools used;
number of overlay statements;
number of assignment statements;
number of IF statements at each nesting level;
number of simple GOTO statements;
number of FOR statements at each nesting level;
number of calls for each procedure and function;
number of exit statements;

COMPILER DATA STRUCTURES

Among the compiler's data structures, two are of fundamental importance to the overall design, and are discussed briefly here in order to highlight their significance. These are the symbol table and the intermediate language (IL).

The symbol table is a collection of entries containing information about all symbols that occur in the source program together with those that are reserved or predefined in the language. Each entry describes all the attributes of a symbol with fields that contain all the declarable or built-in attributes. For a language such as J73, symbol classes include items, tables, blocks, procedures, constant names, define names, zones, keywords, operators and single letters. Attributes of variables include size, range, precision, packing and storage class. Table attributes include dimensions and the ordering and packing of components. Type and constant name attributes include the attributes of their underlying types or values. Constants are represented by both the corresponding type information and their declared or literal value. These values are represented by a single symbol table entry for each value, with a canonical machine independent representation; such a representation is important in allowing accurate constant expression evaluation routines to be used for different target computers.

The intermediate language (IL) is the complete internal representation of the input source program semantics. The IL is initially produced by the syntax analyzer, is modified by the semantic processor and optimizer, and is mapped to target machine language by the code generator. The IL includes codes for all expression operators in the language, for assignment and parameter passing, for all control structures, and for all references to symbols defined in the program.

OPTIMIZATION METHODS

Optimization methods can be divided into two main categories: target-computer-independent and target-computer-dependent. The target-computer-independent optimizations would be concentrated in the optimization module of the typical compiler structure as described in the preceding section. Target-computer-dependent optimizations would be performed primarily in the code generation module.

Machine-independent optimizations are those expressible as transformations of the intermediate language. They are valid (in the sense of not changing the meaning of the program) for all target-machines. There are a number of machine-independent optimizations discussed in the academic literature.[9, 10] Some are applicable to only a few special-purpose programming languages; others can only be performed under certain conditions that rarely occur.

No optimizations can be performed without intimate knowledge of the program being optimized. Data flow analysis is the process by which information is collected for optimization. Data flow analysis can be performed in a number of ways, depending on the nature of the expected source programs.

Traditional data flow analysis algorithms (e.g., iterative analysis and linear nested region analysis[11], as well as more recent high-performance techniques[12], are designed for programs written in Fortran-like programming languages. Such analysis methods are called "low level" because they are preferred for programs that make heavy use of undisiplined GOTO's to express program flow of control. More appropriate for analyzing well-written programs in modern HOLs, such as JOVIAL J73, are "high level" data flow analysis algorithms[13]. Such algorithms are best for analyzing programs that express flow of control with advanced control structures. such as loops, IF, and CASE, with GOTO's appearing only rarely.

Data flow analysis can be either interprocedural or intraprocedural. Intraprocedural analysis collects information from only one procedure at a time, making worst-case assumptions when other procedures are called. The more detailed and time-consuming interprocedural analysis takes into account most or all of the effects of procedures calling other procedures. Fewer worst-case assumptions are made, leading to more opportunities for optimization.

Typical optimizations that result from data flow analysis are:

a. Constant folding (also known as constant expression evaluation).

For example: 1+2 => 3

b. Local and global constant propagation.

```
For example: B=2;              =>      B=2;
             C=1+B;                    C=1+2;
```

c. Local and global common subexpression elimination.

```
For example: B=C*D*E;                  T=C*D;
             F=C*D;            =>      B=T*E;
                                       F=T;
```

d. Re-ordering of expressions, including application of associative and communicative laws.

For example: 2 + B + 1 => 2 + 1 + B

e. Operator strength reduction in loops.

For example:

```
FOR II: A BY B;                T1 = A * 4; T2 = B * 4;
  BEGIN                        FOR II: A BY B;
  ...II * 4...          =>       BEGIN
  END                            ...T1...
                                 T1 = T1 + T2;
                                 END
```

f. Extensive simplification of expressions using algebraic identities.

```
For example: AA * 1 => AA
             AA AND TRUE => AA
```

g. Code motion out of loops (also known as extraction of invariant expressions).

```
For example:
FOR .....;                          T1 = A*B;
  BEGIN                             FOR .....;
  ..... A * B ...          =>         BEGIN
  END                                 ..... T1 .....
<A and B are loop                     END
 invariant>
```

h. Elimination of dead assignments.

For example:

```
  B = C;                            ... <no uses of B>...
...<no uses of B>...        =>      B = D;
  B = D;
```

i. Elimination of unreachable code.

For example: IF FALSE; => <nothing>

j. Optimization of subscript calculations.

For example: A(3,2) the subscript calculation is performed at compile time.

Equally important are target-computer dependent optimizations. There are three basic categories of such optimizations:

- Register allocation
- Code selection
- Generated-code optimization

There is no optimal strategy for assigning intermediate results to computer registers because of the differences between computer architectures. One computer may have index registers, floating registers, and general-purpose registers; another may use general-purpose registers for indexing, but have special registers for multiply and divide. Thus, one register allocation strategy can not work for all target computers. It is possible, however, to extract some basic principles of optimal register allocation:

- It is necessary to keep a full register history during code generation, to avoid redundant loads.
- It is usually optimal to keep values in registers as long as possible, rather than storing intermediate results into memory as soon as they are computed.
- When a register must be chosen for a new value to be loaded, it is necessary to look ahead to the future uses of the value in order to choose an appropriate type of register. Consider the example of a machine that permits addition into both index and general registers, but permits multiplication only into general registers.

  If we wish to compute a sum that will later be used as a multiplicand, it is important that we compute the sum in a general register rather than in an index register. However, if the sum is to be used later as an index, we would want to compute the sum in an index register.
- When all registers are full, it is necessary to have a strategy to determine the "best" register to use (the current contents of the register may have to be saved in memory). Such a strategy could discard the value least recently used or the value least recently loaded. A better strategy is to consider the number of remaining uses of the value, and the distance to the next use; this is called the "usage count" method of register allocation.

Just as no one register allocation strategy can work for all target machines, no one code selection algorithm can be universal. When choosing which instruction codes to use, the code generator must pay attention to special-purpose machine instructions for loop-control, masking, clearing, and incrementing. Halfword, double-word, and multi-word instructions can be used to great advantage when available. Fixed and integer multiplication and exponentiation can frequently be speeded up through use of shift, add, and subtract operations. Exponentiation can be replaced by multiplication when it is profitable to do so. Certain CASE statements can be implemented via branch vectors rather than tests. It is important to make good use of indirect, indexed, and immediate addressing of operands, in order to eliminate useless register loading and redundant allocation of literal storage. A number of algorithms[19] exist that allow the code generator to evaluate a complicated expression in the order that makes minimal use of machine registers.

A number of compilers have demonstrated great success with optimization of the generated machine code. Such optimizations uncover and exploit aspects of juxtapositions of machine instructions that yield optimizations not detectable by earlier optimizers. Many times, such optimizations have produced improvements in unexpected situations, surprising even the compiler's designers. An example of post-compilation optimization is peephole optimization. In this technique, a "window" of only two or three instruction-widths is passed over the generated code. Simple machine-dependent optimizations are sought without examining any code outside of the window. Redundant register loads and stores are eliminated. Operand addressing is improved. Machine condition-codes are used to best advantage. A variety of straightforward but highly effective transformations are performed, cleaning up the rough edges in the code. Other examples of important post-compilation optimizations are eliminating jumps to jumps and reversing the senses of comparisons.

A problem with generated-code optimizations is that they may interact with each other in unexpected and confusing ways, resulting in the possibility of bugs being introduced into the generated code. SofTech has addressed this problem by introducing an inductive proof methodology that demonstrates the correctness of the generated-code optimizer transformations. For each optimization (and a compiler may apply 15 or 20 types of such optimizations), the effects on the contents of memory, the machine condition codes, and the flow of control are explicitly defined. The situations in which the optimization may be performed are also explicitly defined and a proof is constructed showing that the optimization preserves program correctness. It is then possible to construct an inductive proof showing that the aggregate of all optimizations preserves program correctness, regardless of how the optimizations interact.

COMPILER TESTING

Since the compiler is an essential tool in software development, it must work correctly if that development effort is to succeed. It is clearly intolerable for errors to be introduced into the avionics software because of compiler deficiencies. It is only slightly less tolerable for the compiler to fail to successfully compile a valid source program. Because compilers are complex computer programs, they must be thoroughly tested before they are used. Five classes of tests can be defined.

Class I — Tests in this class verify that valid HOL statements are accepted by the compiler. These tests are not intended to verify the correctness of the generated object code.

Class II — These tests verify that the compiler under test rejects statements that are not valid HOL, and that corresponding diagnostic messages are produced when such statements are recognized.

Class III — These tests verify the operation of the implementation-independent HOL directives.

Class IV — Tests in this class are used to check that capacity requirements and constraints are met. For implementation-dependent capacity requirements, model tests are included that can be adapted for a particular implementation.

Class V — These tests verify that HOL programs are translated into correct object code. The tests in this class are all executable and self-checking (i.e., the test program outputs a message telling whether or not the test was passed).

Each class is composed of a great many test programs written in the HOL. The tests are performed by attempting a compilation of each program and, in some cases, execution of the resulting object code. Each test is aimed at exercising a specific language feature or combination of features.

Compartmentalizing the tests in this way achieves three advantages:

a. Catastrophic failure of a single test program will not alter the outcome of subsequent tests.

b. Incorrectly supported language features will have minimal effect on tests of other language features.

c. Partial test sets can be used during compiler development to test partially complete compilers.

Total isolation of language features is not possible since testing of some features must assume that other features are operable. In order to minimize cross-dependencies and to achieve a well-behaved compartmentalization, a nucleus of language features has been defined. Features in the nucleus are tested first. Tests of features outside the nucleus assume the existence of the nucleus features, the existence of the feature under test, and a minimal number of the other language features required to support the feature under test. This organization permits maximum testing support for compilers that are still under development and, also, supports verification of the finished product.

CONCLUSIONS

The approaches to avionics compiler development described in this page have been successfully used by SofTech and others to produce reliable and efficient compilers for avionics applications. The results from the use of these avionics-oriented compilers should convince all but the most hardened sceptics that compilers are an essential part of an avionics software development system. Avionics software development has progressed rapidly in the last five years from relatively primitive support software to integrated and complete software support facilities.

The compiler is at the center of these facilities. The compiler will, in the future, be linked to the simulators and test software so that the benefits of a HOL can extend from the coding phase to debugging and checkout. Through the mechanism of software libraries the compiler will become an information source for software documentation, configuration control, and status accounting tools. Finally, more general and comprehensive software analysis tools (such as data flow analyzers, program structure analyzers, and test instrumentation tools) will be integrated into compiler usage.

Although much greater use of compilers will be made, fewer compilers will be written in the future than the number of applications would indicate. This will be the result of actions taken to minimize language proliferation (e.g., the Ada language), actions taken to reduce computer hardware differences (e.g., the MIL-STD-1750 instruction set), and the establishment of central agencies for compiler certification and distribution.

REFERENCES

1. SofTech, Inc. JOVIAL/J3B Language Specification Extension 2. October 1976. SofTech Document 2051-4.2.

2. MIL-STD-1589A JOVIAL (J73). August 1979. Available from Naval Publications and Forms Center, Philadelphia, PA.

3. Intermetrics, Inc. SPL/I Language Reference Manual. January 1977, Intermetrics Report No. 172-2.

4. Computer Science Corp. Compiler Monitor System-2 User's Manual. June 1969. Navy Contract N00123-67-C-0214 Manual No. M-5012.

5. Intermetrics, Inc. HAL/S Language Specification. 1979. Intermetrics Report No. IR61-10.

6. Woodward, P., P. Wetherall, and B. Gorman, Official Definition of CORAL 66. London: Her Majesty's Stationary Office, 1973.

7. Gessellschaft Für Kernforschung, Full PEARL Language Definition, 1977. Report KFK-PDV 130.

8. Ichbiah, J. et al. "Preliminary Ada Reference Manual," SIGPLAN Notices, Vol. 14, No. 6 (June, 1979).

9. Allen, F. "Program Optimization," in Annual Review in Automatic Programming, Vol. 5. Pergammon Press, 1970.

10. Cocke, J. and J. Schwartz. Programming Languages and Their Compilers. New York: Courant Institute of Mathematical Sciences, 1970.

11. Schaefer, M. A Mathematical Theory of Global Program Optimization. Englewood Cliffs: Prentice Hall, 1973.

12. Graham, S. and M. Wegman. "A Fast and Usually Linear Algorithm for Global Flow Analysis," JACM, Vol. 23, No. 1 (January 1976), pp. 172-202.

13. Babich, W. and M. Jazayeri. "The Method of Attributes for Data Flow Analysis" in two parts. Acta Informatica, Vol. 10, No. 4 (December 1976), pp. 245-272.

14. Sethi, R. and J. Ullman. "The Generation of Optimal Code for Arithmetic Expressions," JACM, Vol. 7, No. 4 (December 1970), pp. 715-728.

SOFTWARE VERIFICATION AND VALIDATION

Donald J. Reifer
President and Chief Scientist
Software Management Consultants
2922 West 227th Street
Torrance, CA 90505 USA

Summary

This chapter defines the terms verification and validation and provides detailed guidance for their conduct. For each activity, it identifies the responsibilities of the participating organizations and discusses applicable concepts, methods, products and problems. Its purpose is to serve as a guide and it should be utilized accordingly.

I. Introduction

The purpose of verification and validation is to provide systematic assurance that software developed for weapons systems will perform its mission requirements economically, efficiently and correctly. This assurance is enhanced by having an objective third-party independently assess the technical adequacy of the delivered software products.

The concept of verification and validation was first employed in early space and missile systems where the consequences of failures were often catastrophic. The concept was extended to encompass nuclear safety analysis and command and control systems and is currently being used on a wide range of systems. Although a quantitative measure of the effectiveness of its application is impossible to make, an examination of the success of past systems employing it indicates that its added expense is justified. The following examples illustrate this point.

The Space and Missile Test Center's verification and validation contractor was tasked to independently evaluate and test a 25,000 word program that had been an integral part of the range safety system at Vandenberg AFB for the previous eight years. Twenty major errors were detected, seven of which were critical to range operation. Possible injury to life and/or property could have occurred if these errors were left uncorrected.

The Minuteman Program Office has employed an independent contractor to do verification and validation for many years. Their overall error history illustrates the benefits attributed to this practice. Minuteman has experienced 1 error per 6000 lines versus an industry average of 1 error per 300 lines. This represents a 20 to 1 improvement.

Other projects such as the Titan missile system, the B-1 Bomber and the Safeguard anti-ballistic missile system have reported like successes. The success of the approach is epitomized by SAMSO Commander's Policy which directs that independent verification and validation will be considered for all space and missile programs employing embedded computer resources.

Verification and validation embody a series of activities which are ideally interfaced with the development process itself. The activities accomplished result in a more orderly and efficient implementation because each development phase produces a verified baseline for the next phase. In addition, errors are typically found early in the cycle before they have a chance to propagate. In summary, the three major payoffs of verification and validation are:

- Improved reliability - fewer errors after acceptance
- Greater visibility - improved chances of success
- Reduced life cost - errors found earlier

Table 1

Verification and Validation Explained

Independent Verification and Validation Is	Independent Verification and Validation Is Not
. An independent technical activity. . Aimed at product evaluation throughout the life cycle. . Identifying errors early. . Employed to insure that all system and subsystem requirements have been fulfilled by the software. . Complementary to the development effort. . Designed to help the developer. . Additional insurance.	. Conducted by the personnel that develop the software. . Checking the code during Development Test and Evaluation (DT&E). . Identifying errors during DT&E. . Employed to insure that only the test requirements of the computer program development specification are met. . A duplication of development activities. . Conducted to harass the developer. . A guarantee of success.

II. Verification and Validation Defined

The terms verification and validation are being used extensively and somewhat interchangeably by members of the software community to describe many disparate testing and analysis activities. Service memos and regulations are often vague and conflicting when discussing the subject matter. The dictionary offers little relief from the confusion because the terms are synonyms for one another. Just what do the terms mean and what activities do they encompass?

Table 1 summarizes what verification and validation is and is not. It is provided to clarify any misconceptions about the processes. For the purpose of this Chapter, verification and validation are conducted by personnel who are not associated with the development organization. This is the key discriminator between verification and validation (V&V) and Development Test and Evaluation (DT&E). The following paragraphs define the terms verification and validation within the context of the acquisition life cycle used by the military.

Verification is defined as the iterative process of determining whether the product of each step of the software development and change process fulfills the requirements levied by the previous step. The four activities that comprise the verification process are briefly described as follows:

- System Specification Verification

  The system-level analytical activity conducted to determine whether the computer-applicable requirements within the system specification represent a clear and accurate translation of the user's need.

- Requirements Verification

  The data system analysis (i.e., hardware and software) activity conducted to determine whether the software requirements reflect the computer-applicable needs denoted by the system specification.

- Design Verification

  The software design analysis activity conducted to determine whether the software design represents a clear, consistent and correct mechanization of the specified requirements.

- Program Verification

  The code analysis and test activity conducted to determine whether the actual code correctly implements the design as described in its associated documentation and whether it is compliant with the specification which contains the design.

Validation encompasses the evaluation, integration and test activities conducted at the system level to ensure that the finally developed software satisfies applicable requirements set down as performance and design criteria in the system specification and/or the software requirements specification.

Successful validation requires that all verification activities are completed. This is necessary because verification procedures often provide a basis for selection of the validation approach.

Validation is usually conducted to ensure system-level requirements are fulfilled. Therefore, software's contribution to performance must be evaluated in a realistic operating environment where hardware, environmental and personnel effects are in the loop.

III. System Specification Verification

System specification verification is the v&v activity conducted to ensure that the system/system segment being considered will meet its mission goals and objectives. Once this activity is completed, the subsystem requirements can be developed in a logical manner with assurance that there is a clear and accurate description of the systems concept.

System specification verification occurs during the validation phase. It takes the system specification and/or data system specification and determines whether the stated requirements are a clear and accurate translation of the user's need.

The validation phase begins with a preliminary system specification and a Test and Evaluation Master Plan (TEMP). The developer's first task is to update the system specification and TEMP so that they are compatible with the approved system engineering concepts and to prepare the System Engineering Management Plan (SEMP). The developer then begins the task of refining the system concept and allocating requirements to subsystems and then to hardware and software configuration items (CI's). The process continues with the developer conducting trade studies which help reduce the risk of the system design. The main products developed during this phase are an authenticated system specification, TEMP, plans, trade studies, preliminary specifications and Interface Control Documents (ICD's).

The IV&V agency is typically brought on contract just after the PO approves the system requirements. Their first major task is to prepare a Verification and Validation Master Plan (VVMP). The IV&V agency then starts a detailed review of the developer's products and reports their findings to the Government's Program Office (PO). The IV&V agency initiates their tool development activity and their test planning during this period. Their participation culminates with their independent confirmation of the feasibility of the requirements.

The PO monitors progress and reviews and approves products produced by both participants. They attend reviews, approve minutes, and assign action items. They work with both the developer and the IV&V agency to provide task direction, establish team spirit and proper working relationships. They review deliverables and evaluate their technical adequacy and acceptability.

Specification verification is concerned with analyzing and evaluating the system specification requirements and their allocations in detail. Detailed requirements analyses are conducted using analytical modeling, simulation, and prototyping to evaluate the proposed conceptual approaches to system mechanization. Preliminary subsystem relationships are reviewed to ensure satisfaction of appropriate performance, functional, and operational requirements. Requirements are segmented in sufficient detail to determine whether the identified design approaches can realize them with acceptable risk.

The IV&V agency should direct their efforts toward evaluating the following three areas during specification verification.

- Risk
- Technical Feasibility
- Supportability

Trade studies are conducted to evaluate alternative system concepts in terms of cost/risk. Typically, the attitude "let the computer do it" prevails. As a result, the cost/risks are not fully evaluated. The IV&V contractor must appraise the Government of the consequences of trades. They must quantify risk in terms of a range of direct (dollars) and indirect costs (schedule). For example, most airborne systems have equation trade studies investigating different guidance or navigation schemes. These trades are the precursor to the derivation of the equations that go in the software requirements specification. Because the equations are the backbone of performance, acceptable engineering solutions (accuracy, speed, etc.) must be verified for a variety of nominal and off-nominal situations. A major change in philosophy could impact hardware selection and software cost. Coding the equations in FORTRAN and executing them with models of the environment using an engineering simulation has proven to be a successful method of proving feasibility early. Other risk reduction techniques include simulation and prototyping. If you've never done it before, it normally pays to build a "quick and dirty" prototype to prove the concept.

Technical feasibility of the functional allocations to hardware, software, firmware, and operator procedures (could be implemented by the pilot) is the next item to be evaluated. The typical philosophy is "let the software do it if it is tricky." With the advent of cheap hardware and firmware, this is not always the right way to go. The IV&V contractor should evaluate the feasibility of the allocations in terms of life cycle costs and appraise the PO of his findings. Analytical modeling can be used to investigate the complexities of real time systems. System simulations which functionally model the architecture can be employed to do hardware/software tradeoffs to assist in allocation. Performance evaluation and workload measurement aids have been used effectively in evaluating performance of existing hardware and software in architectural evaluations.

Another key problem is the tendency of the developer to concentrate on the operational software. Typically, little attention is given to support software used in the development facility, support and test equipment. The availability of critical checkout equipment or a compiler can drive the schedule. The IV&V contractor should ensure that the developer's Computer Program Development Plan (CPDP) adequately addresses these issues. The IV&V agency should spend as much time as necessary (depending on criticality) to ensure that the PO is appraised of the risks and possible consequences in these areas.

Typical problems associated with specification verification activities include (1) overcoming the developer's mistrust of the IV&V agency, (2) expediting information exchange, (3) evaluating fully the consequences of trades and (4) involving software personnel in interdisciplinary working groups so that they are not at the mercy of other technical disciplines.

IV. Software Requirements Verification

Requirements verification is the v&v activity conducted to ensure that the software requirements can accomplish their allocated system requirements. Its primary aim is to identify ambiguous, ill-defined and technically inadequate software performance and design requirements as early in the process as possible.

Requirements verification occurs during the validation phase. It ensures that the computer program development specifications adequately reflect the computer-applicable portion of the system specification. The major software product of this activity is a set of authenticated specifications which become the allocated baseline for the Full Scale Development Phase.

The developer's responsibilities are to (1) revise his software requirements specifications and ICD's based upon continuing requirements definition activities and (2) support the conduct of a software requirements review. The requirements specified should be finalized when they are sufficient to form an allocated baseline for design. The requirements should then be reviewed at a software requirements review where an action plan for approval of the software requirements specifications and subsystem ICD's should be formulated. The approved software requirements specifications and subsystem ICD's will form the basis of Full Scale Development.

The IV&V agency's responsibility during this period is to evaluate the developer's products to ensure their technical viability with regard to the computer-applicable requirements of the system specification. Requirements are analyzed and are sometimes independently derived in order to verify the developer's allocations which form the basis of design. The IV&V agency is as responsible for the requirements as the developer. They must do everything necessary to give the PO their assurance that the software requirements specifications and other supporting documents are technically sound.

The PO continues to monitor progress and review and approve products produced by both participants. The PO attends reviews, chairs working group meetings, institutes technical interchange meetings, approves minutes and assigns action items. The PO's major responsibility is to make sure that the requirements get defined and specified in a form appropriate for baselining. The PO must also make sure that the schedules are maintained for support and checkout equipment needed for software production. If the requirements in the software requirements specification are ill-defined, the PO should extend the definition activity. In making their decision, the PO must listen to both the developer and the IV&V agency. Baselining too soon can lead to large cost overruns. Baselining too late could cause delays and other problems.

Requirements verification is concerned with evaluating the developer's [illegible] phase products in detail in order to confirm that they form an appropriate [illegible] the Full Scale Development Phase. The software requirements specifications [illegible] to ensure their requirements are consistent, complete, testable, [illegible] appropriate. The evaluation is directed towards answering the questions [illegible] evaluation tools and techniques employed to answer these questions [illegible], simulation, algorithm evaluation testing, requirements [illegible] reviews.

Table 2

Software Requirements Verification Checklist

- Are all functional, interface, and test requirements completely specified in quantitative terms?
- Are there any potential problem areas in fulfilling the requirements?
- Are the requirements logical, consistent, testable, traceable, and understandable?
- Are the requirements sufficient to realize both the system and subsystem objectives?
- Are all input, output, and processing requirements identified and specified for each function without ambiguity?
- Are all hardware and software interfaces identified?
- Are the data base and data requirements clearly stated?
- Are acceptance criteria specified for each requirement?
- Have the equations been scientifically verified?
- Have the human engineering aspects been addressed adequately?
- Is there early and continued emphasis on test planning?
- Are the objectives and stages of testing described?
- Do timing and sizing estimates have sufficient margin?

The IV&V agency should direct their efforts towards evaluating the following four areas during requirements verification:

- Technical Adequacy
- Criticality
- Testability and Supportability
- Timing and Sizing

The primary objective of requirements verification is to confirm the technical adequacy of the requirements. The specifications are first evaluated for completeness, consistency, and traceability to the system specification. Special requirements language systems have been developed to effectively automate part of this process. Then, the detailed functional and performance requirements are analyzed in great detail. Some IV&V approaches that have proved successful in the past include:

- Use of scientific simulations ennanced with more sophisticated models (i.e., sensors, vehicle, atmospheric, etc.) to verify the accuracy of the equations in their engineering form for realistic environments.
- Use of functional simulations to evaluate interrelationships between functions and functional performance (i.e., timing, sequencing, etc.)
- Use of protypes to validate requirements derived for functions for which little or no history exists (e.g., a new redundancy management technique).
- Use of capability matrices or N^2 charts to trace functions or their interfaces vs. other requirements.

An essential part of functional analysis is determination of criticality. In many instances, the cost of IV&V prohibits its cost effective application to the entire program. Only those functions deemed critical, then, are subjected to an IV&V. Candidates for IV&V could include such functions as a terminal guidance function for a homing interceptor, a range safety destruct function, a critical bombing algorithm, a precision radar tracking function on an aircraft, and an entire flight program for a launch vehicle.

Another area of concern revolves around the tendency of the developer to concentrate on performance at the expense of testability and supportability. The IV&V agency should ensure, using analyses, that every requirement stated in the specification is testable. This requires a detailed examination of the TEMP and test requirements section of the software requirements specification. Having the IV&V agency review test documentation is controversial. One school says they shouldn't because it will bias the IV&V test approach. Another school says they should because only then can the IV&V program be made complementary to the developer's approach. In addition, the specifications should be evaluated to ensure they are consistent and compatible with the provisions of the PO produced Computer Resources Integrated Support Plan (CRISP). Supportability is as important a consideration as testability.

The final areas of concentration for the IV&V agency is that of timing and sizing. The IV&V should independently derive timing and sizing estimates based upon their experience. These estimates can then be compared with the developer's and disparities should

be examined before budgets are established.

Typical problems associated with requirements verification include (1) continuing to maintain the teamwork and information exchange, (2) overcoming the continual pressure to prematurely baseline the requirements especially when their technical adequacy is questionable, (3) maintaining schedule when critical decisions that impact software or products from other disciplines are delayed and (4) ensuring a realistic test approach is developed in parallel with the software requirements specification activity.

V. Software Design Verification

Design verification is the v&v activity conducted to ensure that the software design represents a clear, consistent, and accurate translation of the software requirements. Its primary aim is to confirm the fact that the recommended design will do the job specified in the software requirements specification. It does not attempt to redesign. Instead, it seeks to identify inadequacies in the design and test approach before implementation starts.

Design verification is a Full Scale Engineering Development Phase activity. It takes the General Design Specification in two versions (preliminary and detailed) and ensures that the evolving design adequately satisfies the provisions of the software requirements specification. The major product of this activity is a set of specifications which are detailed enough to form the basis of coding.

The developer's responsibilities are to (1) formulate general software design and test concept, (2) develop a detailed design using this concept that fulfills the requirement of the software requirements specification and (3) support the conduct of a Preliminary Design Review (PDR) and a Critical Design Review (CDR). The Full Scale Engineering Development Phase should start with authenticated software requirements specifications and ICD's. These should be updated and an acceptable design and test approach should be developed that meets their intent. The PDR should provide an action plan for approving the approach which establishes the design architecture for each software CI. This architecture is then refined successively until it is of sufficient detail to commence coding. A CDR is then held to provide an action plan to finalize the design and test procedures. The CDR data package typically consists of an agenda, General Design Specifications, draft test procedures, draft users manual, and draft version description documents.

The IV&V agency's responsibility during this period is to evaluate the developer's products to ensure their technical viability and to contribute to the design refinement process. The design is checked for logical consistency and completeness. Key algorithms may be either simulated or rederived in order to assess their technical adequacy. The IV&V agency must do as much analysis as is necessary to independently verify the design implementation. They provide the PO with their assurance that the design is technically sound and that its critical components will do the job.

The PO continues to monitor progress and reviews and approves products produced by both participants. The PO attends reviews, chairs working group meetings, institutes technical interchange meetings, attends design inspections, approves minutes, and assigns action items. Their major responsibility is to make sure the design is finalized by CDR. The PO must also make sure that all the supporting checkout and production equipment is available once the decision is made to go ahead with coding. They may wish to use an incremental development approach and hold several CDR's to preserve the schedule.

Design verification is concerned with evaluating the software design in detail in order to confirm that it serves as an appropriate baseline for coding. The General Design Specifications are evaluated to ensure their provisions are both consistent with the software requirements specifications and adequate to do the users processing job. The evaluation is directed towards answering the questions posed in Table 3. Evaluation tools and techniques employed to answer these questions include simulation, prototypes, design languages, walkthrus, design inspections, process construction, and performance evaluation.

The IV&V agency should direct their effort towards evaluating the following four areas during design verification:

- Technical Adequacy/Performance
- Modularity and Maintainability
- Timing and Sizing
- Support Equipment Availability

The primary objective of design verification is to confirm the technical adequacy of the design. The total software design must be expressed in writing, simulated, analyzed, and evaluated as to risk, expected performance, cost, and reliability. The evaluation must consider performance capabilities, system and software architecture, operational sequences, information flow, timing, scenario design and many other parameters. Some IV&V approaches that have proven successful in the past include:

Use of design languages to incrementally document the design.

- Use of discrete event architectural simulations to assist in making key design decisions relative to intermodule sequencing, control laws, communications processing and/or executive structure.
- Use of trial coding to confirm the performance or resource consumption of critical modules (identified during requirements verification) in a typical operating environment under nominal and stress conditions.
- Use of rederivation of key algorithms to assure optimality and to understand assumptions and approximations.
- Use of dimensional analysis to evaluate algorithms and data for completeness and compatibility.

Table 3

Software Design Verification Checklist

- Have all software requirements been addressed in the design and is there traceability?
- Are all the equations, algorithms, and input/outputs correct?
- Is the data base fully defined and is its architecture (structure and access methods) fully compatible with the logical design?
- Are the specific module capabilities and their complex control and data linkages defined?
- Are the inter-module communications and interface rules established in the software requirements specification fully adhered to in the design?
- Is the design compatible with the hardware and software interfaces established in ICD's and the software requirements specifications?
- Does the design reflect the current version of the requirements (includes all changes)?
- Are there timing and sizing budgets established at the module level?
- Are the test procedures compatible with the design, test plan and software requirements specification test requirements?
- Do the individual designs fully realize overall requirements for performance, operation, growth, maintainability, etc.?
- Is the design detailed enough to begin coding?
- Is there sufficient timing and sizing margin at CDR?

Design verification activities must also ensure that the design is modular and maintainable. Software should be designed to accommodate change. The design should be evaluated to make sure the modularity rules (e.g., minimize intermodule communications using the Parnas information-hiding principle), testability and maintainability considerations are embedded within its structure. These provisions cannot be implemented as an afterthought. They must be an integral part of the design or else they will fail to be effective.

The next area of concentration for the IV&V agency is their timing and sizing analysis. The IV&V agency should continue to refine their estimates and compare them with those derived by the developer. The resulting budgets will be more realistic as a result.

The final area of concern is that of support equipment availability. The IV&V agency should assist the PO by monitoring the developer to ensure that the support software (e.g., compilers, simulators, etc.) and checkout equipment that is needed to start coding is available at the CDR. The IV&V agency must also police itself and assure that its tooling is available and qualified as well.

Typical problems associated with design verification include (1) continuing to maintain the teamwork despite petty disputes, (2) overcoming the continuing pressure to prematurely baseline the design even though it is not modular, incompatible with the machine selected and/or based on volatile requirements, and (3) ensuring that the design is testable and that the user is involved during the design process.

VI. Program Verification

Program verification is the v&v activity conducted to independently assure that the actual code that is developed is compliant with the technical description contained within the approved design specification. Program verification is that activity that ensures sanity, evaluates sequencing logic, file structuring, execution paths and limitations, and interfaces to name a few. This activity does not, however, evaluate the program's performance in a real or pseudo-real environment. That is the task of validation.

Program verification is a Full Scale Engineering Development Phase Activity. It takes the code as it is produced and compares it with the design specifications against which it was generated. It works with the object and source code. It is usually scoped to complement the developer's DT&E activities, not to duplicate them. Program verification is not a DT&E or a software integration activity. It may employ DT&E methods, but its aim is different. It is a separate and independent activity directed towards providing the PO with additional assurance that the code will properly realize the design. The output of this activity is code that fulfills its specifications.

The developer's responsibilities during the period starting with the CDR and ending with the Final Qualification Testing (FQT) are to (1) code and checkout the individual software modules, (2) integrate the modules into software CI's, (3) conduct successful Preliminary Qualification Tests (PQT's) and Final Qualification Tests (FQT's) for all software CI's, (4) support the conduct of formal audits, and (5) support the conduct of a Test Readiness Review. The developer starts with the approved design specifications and implements them. Implementation can be accomplished using a top-down (i.e., build-a-little and test-a-little), bottom-up or alternative methodology (e.g., hardest-out-first). Each module developed is tested stand-alone and in combination with other modules. Integration tests for the software CI is then accomplished using regression, string, or other testing approaches. Finally, FQT's are conducted and audits are held. FQT's are formal tests of the integrated software CI, performed by the developer and witnessed by the PO, conducted to demonstrate that the software CI fulfills its requirements. They differ from PQT's in the following areas:

- PQT's are normally much more detailed in terms of coverage.
- PQT's normally provide only minimal hardware/software interface testing.
- PQT's are normally conducted at the contractor site using simulated equipment and environments.

The IV&V agency's responsibility during this period is to independently test and evaluate the developer's product(s) using his own facilities and tools. The code is checked for errors, omissions, and incorrect translations using a variety of methods during production. The IV&V agency must do as much analysis as necessary to verify that the code correctly implements the design. The IV&V activity differs from the developer's DT&E tasks in the following areas:

- Program verification is conducted against the General Design Specification rather than the software requirements specification.
- Program verification is usually less formal and less structured than either PQT or FQT.
- Program verification is usually more stress oriented than PQT.
- Program verification is conducted to discover and correct programming errors, not to confirm proper implementation (a major philosophical difference).

While program verification looks at design, validation may look at software requirements in addition to system specification needs. This Chapter clarifies the distinction in roles for the reader in the next section.

The PO again monitors progress and reviews and approves products produced by both participants. The PO attends reviews, chairs working group meetings, institutes technical interchange meetings, resolves discrepancies, approves changes to specifications, approves minutes, and assigns action items. They conduct audits (both formal and informal) during this critical period to assess progress and confirm that the product that underwent test and that delivered are one in the same. They observe test conduct and analyze test results.

Program verification is concerned with providing confirmation that the code fulfills the requirements of the General Design Specification. Confirmation is accomplished by addressing the questions listed in Table 4. Tools and techniques employed to answer the questions posed by the checklist include automated test generators, comparators, cross-assemblers, data analyzers, decompilers, debug aids, dynamic analyzers, dynamic simulators, editors, emulators, flow charters, etc. Most so called v&v tools address this activity. They have been developed in many cases to help perform unit, module, subsystem, and integration testing. These tools analyze the code in detail to determine whether there are errors present.

The IV&V agency should direct their efforts towards evaluating the following three areas during program verification:

- Technical Correctness
- Efficiency
- Technical Adequacy

The primary objective of program verification is technical correctness. The actual program code in its source and object form is evaluated against its design specification and discrepancies such as those listed below are identified for correction:

- Incorrect logic flow
- Inaccuracies in mathematical calculations
- Incompatible interfaces
- Improper use of instructions

Some IV&V approaches that have proven successful in identifying these and other errors in the past include:

- Use a verification approach that combines the virtures of functional, logical and path testing.
- Concentrate your effort on the interfaces and sequencing logic. Statistics show these areas to be very error-prone.
- Perform both static and dynamic execution analysis of the code. Static analysis will scrutinize the code and execution analysis will scrutinize the results.
- Use tools and approaches that allow for test repeatability and variable fidelity.

Table 4

Program Verification Checklist

- Has every software module been checked to determine whether it produces correct output for prescribed inputs?
- Are the arithmetic results correct for nominal conditions?
- Are singularities and other conditional occurrences of data processed correctly?
- Are the subroutine calls properly formatted and has each been tested?
- Are the parameters dimensionally correct and is their calling sequence properly invoked?
- Is scaling proper to realize correct precision and desired results?
- Have all error conditions been processed correctly?
- Have all instructions and each branch been exercised at least once?
- Have the timing and resource allocations been properly mechanized?
- Is the task sequencing proper to mechanize the function in correct execution order?
- Is the compiler producing acceptable code?
- Are there any violations of agreed-upon programming practices?
- Is the users and program description documentation adequate?

Program verification also addresses the efficiency problem. The program is continuously monitored as it is being developed to insure that timing and sizing budgets established during design are met. Detailed module timing analyses are conducted to identify modules that are marginal in processing data within prescribed time limits. Size is monitored. A key problem that typically causes size and timing growth is compiler inefficiency. The target computer code generator usually requires modifications to its optimization techniques even in the best of circumstances. The use of floating-point instructions in excess of what is thought to be an optimal mix for the intended application is another problem area.

The final area of concern is the technical adequacy of the code and related software products. Program verification ensures that the code is fully and correctly described in the detailed General Design Specification which also serves as as-built documentation. The General Design Specification should describe the program, not some lesser version of it. Program verification is also concerned with ensuring that the Users Manual is adequate. Lastly, program verification is concerned with assuring that the documentation adequately tracks the latest versions of the code.

Typical problems associated with program verification include (1) resolving resource utilization problems and conflicts, (2) overcoming mechanization problems on the target computer, (3) recovering from late hardware deliveries, (4) overcoming problems associated with unreliable hardware, (5) compensating for requirements changes, and (6) ensuring that nominal test results are complemented using stress tests.

VII. Software Validation

Software validation is the v&v activity concerned with determining whether all software and system performance, interface, functional and test requirements are being satisfactorily fulfilled. Software validation is that activity that ensures that every requirement is adequately tested and that the software has been adequately shaken down from a system perspective. Unlike program verification, validation seeks to evaluate the program's performance in a real or pseudo-real environment.

Software validation is a Full Scale Engineering Development Phase activity normally conducted somewhat in parallel with program verification. It takes the code as it is produced and compares it with the System Specification and software requirements specification against which it was generated. It works with both the source and object code. It differs from program verification in purpose and in detail. Validation usually involves operational exercise of the code to assure that the requirements are met, while program verification usually involves the detailed analysis required to verify the design's proper implementation. In some instances, software validation activities overlap those conducted by the developer in the area of DT&E. The IV&V agency is tasked with providing a second opinion on the software's ability to perform. The IV&V agency will test those critical functions identified during system specification and software requirements verification to provide the evidence he needs to confirm the software's capabilities. If the entire program is critical, the IV&V will run a totally independent DT&E to qualify it from their vantage point. The output of this activity is code that fulfills system level requirements.

The developer's responsibilities during the period starting with the FQT and ending at the System Readiness Review are to (1) integrate the software CI's with other software CI's and the hardware, (2) conduct system level tests, and (3) support the conduct of formal reviews and audits and IOT&E. The developer starts with all qualified hardware and software CI's. He integrates them together and tests the composite system in accordance with the provisions of the TEMP. In some instances, the system is transitioned to a Government team which conducts IOT&E of the integrated system before it is deployed.

The IV&V agency's responsibilities begin earlier in the life cycle. They test and evaluate the code that was identified as critical in parallel with both its code verification and the developer's DT&E activities. Their job is to provide feedback early enough so that problems identified can be corrected without costly schedule impacts. The IV&V agency accomplishes its job by providing independently derived test results against which the developer's results can be compared. The IV&V agency also actively participates in the FQT and the System Readiness Review. The formal results of the IV&V agency's test and evaluation efforts are presented at these reviews.

The PO continues to monitor progress and review and approve products produced by both participants. The PO participates and witnesses test conduct and analyzes test results. They chair working group meetings where both the developer and the IV&V agency present the results of their testing. They resolve problems and act as the arbitrator for disputes. They attend reviews, chair technical interchange meetings, approve changes to the specifications, approve minutes, and assign action items.

Programs are validated to confirm that they perform in accordance with their system and software requirements. Confirmation is accomplished by executing the completed code in a realistic environment according to the following three stage approach:

- Software CI Testing
- Integrated Software Testing
- System Testing

Software CI testing is that formal testing conducted to confirm that each and every requirement of the approved software requirements specification has been fulfilled. Software CI testing is accomplished by the developer, witnessed by the PO and independently evaluated by the IV&V agency (if warranted). It involves both PQT and FQT. It can be achieved incrementally in either a top-down or bottom-up fashion. It uses approved DT&E procedurs which are compatible with the test plan approved for demonstrating the software requirements. A checklist for the conduct of software CI testing is illustrated in Table 5.

Table 5

Software Testing Checklist

- Are all inputs accepted and all outputs produced?
- Does the mechanization of algorithms and models fulfill the prescribed requirements?
- Can the function being performed by the module be exercised at the extremes of the range of input variables?
- Are the initialization provisions properly implemented?
- Are the error handling provisions properly mechanized and has every error condition been tested?

The following problems should be addressed by all parties that participate in software testing:

- Designing an effective set of test cases.
- Creating an efficient test environment.
- Managing test data.
- Knowing when to stop testing.

Methods that have been used effectively to attack these problems include:

- Designing test cases against established test criteria similar to those listed in Table 6.
- Designing test cases that exercise software capabilities, not features.
- Using test tools effectively to create an efficient diagnostic environment.
- Creating a test data base that relates each test to its requirements and manages test cases and test results.
- Setting pre-defined, realistic goals against which test accomplishment can be measured.

Table 6

Example Test Criteria

. Programmer judgement
. Execution of all program statements
. Execution of all program branches
. Dividing program paths into equivalence classes and executing at least one path from each class
. Execution of randomly-selected test data
. Execution of all legal program paths
. Stress test at boundaries

The IV&V agency's involvement in testing is dependent on their assessment of criticality. For non-critical cases, they should participate as an independent reviewer of the developer's products. Test documentation should be reviewed with a concentration on procedures and results. The procedures should be reviewed to ensure:

- The test procedure tests the program and not a simpler variation of it.
- There is positive feedback in every test procedure.
- The results of the test procedure are not only predictable, but predicted.
- Test results meet all acceptance criteria.

Test results should be evaluated against expectations, previous results, and requirements. Results should be further examined to determine if the test objectives established have been realized. Criteria established in the test plan and software requirements specification serve as guides to this determination.

The IV&V agency should independently test those software CI's designated as critical. Test plans and procedures should be developed to define what tests will be conducted to evaluate the program. Care should be exercised to ensure that the test program is independent of the developer. The major differences between the IV&V and the developers test program are as follows:

- The developer's test program is much more formal. PQT's and FQT's are conducted for each software CI. The IV&V agency's test program is usually less structured.
- The IV&V agency's concentration is more test oriented because that is their one and only job.
- The developer stresses functional testing at the expense of logical, path and stress testing. The IV&V agency normally gives equal attention to all four techniques.

Integrated software testing is accomplished to validate the overall operation of the data system as an integrated entity in a pseudo-operational environment. The software CI's are integrated together and with the hardware and tested to ensure that the provisions of the system or system segment specification are fulfilled. The common myth that the developer's software organization's job ends with validated software CI's is disspelled.

The software developer participates as an essential member of the test team. As integration and test progresses, software personnel accomplish the following tasks:

- Review test results and identify problems.
- Review problem solution approaches for potential software impacts.
- Modify qualified software CI's.
- Retest modified software CI's.
- Modify software requirements specifications and related documentation.
- Advise the Test Director of the problems and pitfalls associated with solving all the world's problems through software.

The IV&V agency supports integrated software testing and contributes directly as a member of the team. They provide independent impact assessments and reverify and revalidated modified software CI's as required by a needs assessment.

Finally, the integrated system is validated in an operational environment and turnover and transitioning is accomplished. System validation is accomplished to ensure that all the provisions of the System Specification are fulfilled. The following planning documents could play an important role in system testing:

- Test and Evaluation Objectives Annex (TEOA)
- Test and Evaluation Master Plan (TEMP)
- Computer Resources Integrated Support Plan (CRISP)
- Program Management Responsibility Transfer (PMRT) Agreements
- Turnover Agreements

System testing is normally conducted by a government team with assistance provided as needed by the developing contractor. The IV&V agency may participate as a team member.

Typical problems associated with validation include (1) ensuring that sufficient time is allocated to system test so that the system is not prematurely accepted and fielded, (2) ensuring sufficient software manpower is committed to support system test, (3) providing backup test facilities to preserve schedules when continual hardware reliability problems occur, and (4) resolving function caused by integration disputes.

VIII. Conclusions

The definitions and description of the activities encompassed by them are offered to provide guidance to those seeking to create a technically sound assurance technology for computer software. As we have described in the Chapter, verification and validation can be employed to independently monitor and evaluate the development of software throughout its life cycle. The concepts embodied by this technology are not new. They have and are being used to improve the quality of modern weapons systems. Yet their employment requires careful planning and tailoring to the peculiarities and constraints of the individual project. Those interested in material relative to planning and tailoring are referred to the references which serve as an extension to this Chapter.

IX. References

1. D. J. Reifer, The Aerospace Corporation, Computer Program Verification/Validation/Certification, 1974, Report No. TOR-0074(4112)-5.
2. D. J. Reifer, The Aerospace Corporation, and Lt. R. L. Ettenger, U. S. Air Force, Test Tools: Are They A Cure-All?, 1974, Report No. SAMSO-TR-75-13.
3. D. J. Reifer, The Aerospace Corporation, "Automated Aids For Reliable Software", Proceedings of the International Conference on Reliable Software, 1975, available from the IEEE.
4. D. J. Reifer, The Aerospace Corporation, A New Assurance Technology for Computer Software, 1975, Report No. SAMSO-TR-75-238.
5. D. J. Reifer, The Aerospace Corporation, Microprogram Verification and Validation, 1976, Report No. SAMSO-TR-76-217.
6. D. J. Reifer, The Aerospace Corporation, "Computer Program Verification and Validation", Proceedings of the Invitational DOD/Industry Conference on Software Verification and Validation, 1976.
7. D. J. Reifer and S. Trattner, The Aerospace Corporation, "A Glossary of Software Tools and Techniques", Computer, Vol. 10, No. 7, July 1977.

8. D. J. Reifer, The Aerospace Corporation, "The Software Engineering Checklist", Proceedings of the AIAA/IEEE/NASA Computers in Aerospace Conference, 1977.

9. D. J. Reifer, TRW, Verification, Validation and Certification: A Software Acquisition Guidebook, 1978, Report No. TRW-SS-78-05.

10. D. J. Reifer, TRW, "Software Quality Assurance Tools and Techniques", Software Quality Management, New York, Petrocelli, 1979.

SOFTWARE MAINTENANCE MANAGEMENT PROCESS

by
William R. Bogdan
Tactical Mission Software Branch Head
Fleet Software Engineering/Analysis Division
Software and Computer Directorate
Code 504
Naval Air Development Center
Warminister, Pennsylvania 18974
United States of America

SUMMARY

This paper deals with the management concepts utilized in developing a maintenance capability to support fleet tactical software. The discussion begins with a brief description of the tactical software being maintained. The kind of management planning that is essential to fleet software maintenance is then presented. Software design factors are discussed that will reduce maintenance costs by making the initial software development more amenable to subsequent modification. The methodology the Navy uses to control changes to a software configuration baseline is addressed and how the Navy insures reliability of the software prior to fleet release is included. Also provided is an overview for the work breakdown structure and the organization of resources for work accomplishment. The paper concludes with a discussion on estimating procedures for determining software maintenance cost. Note that the concepts addressed in this paper on the management process for software maintenance are widely used by Navy agencies but not precisely as given herein.

SECTION 1. INTRODUCTION

Software maintenance can be defined as the development and implementation of modifications to software after the software is available for operational use. The modifications are made to correct faults and to provide improved or modified functional capability. Most maintenance performed on software is for the latter reason. The initial intended use of the software changes after the software system is released for operations, perhaps because users see better ways to accomplish their tasks, or new and additional demands must be processed by the system. It is the changes in the functional requirements of the system that software maintenance accomplishes by implementing modifications to the existing software.

Software maintenance is just as challenging and difficult as the development of new software. It requires the same steps and attention to detail as in a software development. The implementation of a modification to software must include definition of functional requirements, design, code, debug, integration, and acceptance testing. In some ways the software maintenance can be more difficult in that the existing memory and timing reserve will not permit an expansion or improvement of the functional capability of the software. This forces the software maintenance engineer to develop unique design improvements to the software architecture to increase the efficiency of the use of memory and processing time.

The purpose of this paper is to describe Navy management considerations and concepts which are utilized in developing engineering support for the maintenance of Navy tactical software. It is these considerations and concepts which constitute the management process for software maintenance. This paper will address some of the cradle-to-grave aspects of life cycle support of software, but with particular emphasis on maintenance requirements after the software becomes operational. Note that the management process discussed herein is for the maintenance of large system, real time application software developed for operations in the fleet.

The management process for the maintenance of tactical software involves decisions in establishing control of changes to the software and in providing for the implementation of improved functional capability throughout the life cycle of the software. It is important that the planning and operations of the software maintenance be thoroughly conceived and established early in order to continually deliver software to the fleet that has high functional effectiveness and high reliability. Essentially, the management process is a group of considerations and concepts that management includes in their planning deliberations and incorporates into their approach to providing for the maintenance of tactical software. These considerations and concepts are as follows:

a. Life Cycle Support
b. Early planning
c. Identification of the software management organization
d. Software design factors that will facilitate the maintenance of software
e. Configuration Management required to control software change
f. Quality Assurance to insure software reliability
g. Work breakdown and staffing
h. Maintenance cost

Each of the above concepts will be presented in further detail in the ensuing discussion and in the order shown. However, to improve the reader's appreciation to the importance and necessity of the above management concerns, a brief description of the kind of software being maintained will be given first.

SECTION 2. SOFTWARE DESCRIPTION

The software being maintained is fleet tactical software which includes some of the largest and most complex real time application software in existence today. Tactical software provides the control of electronic sensors and displays, computation of tactical and navigational algorithms, disposition and execution of ordnance and armaments, and the real time computer response to man-machine interface demands. These automated tasks are performed by the software for the purpose of accomplishing mission requirements and tactically pursuing targets of Navy interest.

Tactical software for a given application or system is a large conglomerate of software. Less than one-third is used directly in the mission. The major portion of this software is used for system test, code generation, simulation and training. In order to facilitate the Navy's management of this software complex, the software is generally divided into the following major systems:

a. Tactical Mission Software
b. System Test Software
c. Support Software
d. Trainer Software

Tactical Mission Software. Tactical Mission Software provides for the control of system tactical equipment and the capability for processing, storing, correlating and displaying tactical data. This enables the crew officers and operators to optimize the utilization of mission tactics and the system equipment. System equipment may consist of electronic sensor hardware such as acoustic signal analyzers, radar, infrared detection and electronic emission detection. Other equipments that may be included in a tactical system are magnetic tape transports, drum and disk storage, printers, keysets, cathode-ray-tube displays, data link communication receivers/transmitters, and various types of navigation equipment such as inertial, doppler and Omega. Tactical Mission Software varies in size as high as 300,000 core words and is written for the most part in high level languages and is executed on the tactical system's militarized computers.

System Test Software. System Test Software provides system "Go-Nogo" tests which enable the crew members to verify operational readiness. In the event a fault is encountered during the "Go-Nogo" testing, the System Test Software provides diagnostic software for fault isolation down to a hardware circuit board. System Test Software varies in size up to 900,000 core words and is written in assembly language and high level language. Assembly language is used extensively to permit bit structures to be exercised and analyzed. System Test Software is also executed on the tactical system's militarized computers.

Support Software. Support Software is executed on laboratory facility computers. This class of software is comprised of compilers, assemblers, loaders, code generators, simulation software, data reduction programs and utility routines.

Trainer Software. Trainer Software consists of system simulation software and a modification of the Tactical Mission Software. Trainer software is resident on a replica of the tactical compartment and is used to train crew members to perform their particular operator functions and to perform as a cohesive unit. Note that the simulation software is used to simulate mission environment at the Trainer site and to simulate some of the system hardware that would be difficult or impractical to include within a Trainer tactical compartment.

SECTION 3. LIFE CYCLE SUPPORT

The planning to acquire and implement resources for software maintenance must:

a. consider the entire life of the software,
b. begin early in life of the software in order to reserve funding and identify sufficient personnel resources for outyears,
c. define early who is responsible for the maintenance.

A concept which focuses and organizes management's thinking on these planning considerations is Life Cycle Support. Navy Life Cycle Support of software is an integrated cradle-to-grave process. This forces software developers to resolve, early in the software design stage, the requirements for fleet logistic support (i.e., maintenance support) of the software. For the purpose of providing an orderly approach to the Life Cycle Support of software, the Navy divides the life of the software into three phases as shown in Figure 1.

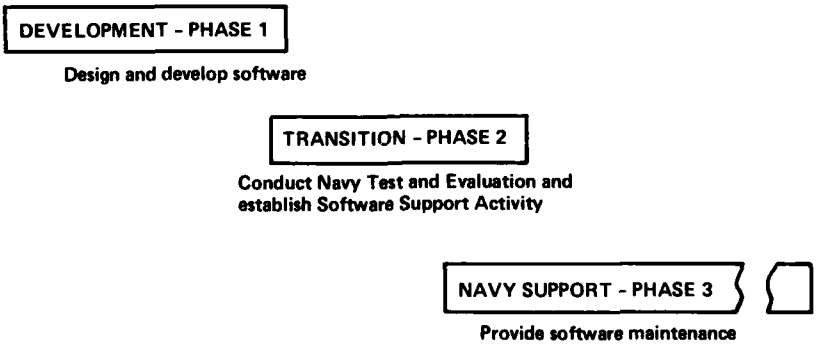


FIGURE 1 – Software Life Cycle

Development Phase. The Development Phase includes those activities which cover the requirements definition, design, code, debug, integration, and test of a new software system. As part of the early planning for a software system, Navy agencies or commands are selected to assume responsibility for each phase of the life cycle. The agency assigned as the Development Activity assumes responsibility for the software system's Phase 1 - Development. The agency assigned responsibility for maintenance of the software is called the Software Support Activity. The Software Support Activity assumes responsibility for Phase 3 - Navy Support. Both the Development Activity and the Software Support Activity share responsibilities for Phase 2 - Transition.

Transition Phase. The Transition Phase includes a series of events that are directed toward the assumption of custody of the software by the fleet and the development of the capability to maintain the software. During the Transition Phase, the Navy conducts formal Test and Evaluation in order to validate that the software will satisfy fleet operational requirements. At the same time, the Software Support Activity begins its build-up of personnel, establishment of facilities for software development and test, training of personnel, and assumption of the Configuration Management.

Navy Support Phase. The Navy Support Phase is the software maintenance phase and it is key to the Navy's ability for maintaining the readiness of its software in the face of changing threat conditions. This phase commences at the time the software is released to the fleet for operational use and remains in effect for the balance of the life of the software. During this phase, fleet users submit requests to the Software Support Activity for software modifications which result in periodic updates and re-issues of functionally improved software.

The Navy Support Phase minimizes the obsolescence of tactical systems. Because these systems are programmable, it is possible to recode portions of the software and create new functional capability. In their use of the system, fleet operators and officers determine and suggest many unique functional improvements to the operation of the tactical system. Many of these improvements can be implemented via the software or as a software work-around instead of hardware change. Periodically the Navy will make major hardware revisions and/or additions to a tactical system; however, these hardware updates do not occur frequently. Through periodic software re-issues (occurring approximately 12-18 months apart) during the software maintenance phase, the fleet can obtain significant system improvements and thereby increase tactical proficiency. The re-issue of the software with significant improvements permits effective resolve of the problem of changing threat conditions that the fleet constantly faces.

It should be clear to the reader at this point that software maintenance as addressed in this paper, refers primarily to the implementation of software functional improvements. Software problems or fault corrections are undertaken during the Navy Support Phase, but fault corrections are a minor concern within Navy software maintenance efforts. The reason for the minor concern is that extensive testing performed on software prior to release to the fleet minimizes the occurrence of software faults after release.

Early Planning Requirements. Because fleet tactical software has a relatively long life consisting of numerous changes in functional requirements and because outyear costs are heavily dependent upon management decisions made during the development phase of software, life cycle planning must be performed at the beginning of the software development process. The Navy has found it useful as part of the development of new software to require software managers to develop this early planning which is documented and referred to as the Software Life Cycle Management Plan. The Software Life Cycle Management Plan is high level planning which addresses critical cradle-to-grave management concerns for a particular software system. The following is a brief outline of the contents of the Software Life Cycle Management Plan document:

Volume 1:

Sect 1.0 – States the purpose, objective, and scope of the plan.
Sect 2.0 – States applicable standards and instructions that the software development and maintenance will adhere to.
Sect 3.0 – Identifies or briefly describes the software being managed.
Sect 4.0 – Presents the life cycle schedule, milestones, potential risks, and contingency plans.
Sect 5.0 – Establishes who is in charge/responsible.
Sect 6.0 – States how Configuration Management will be pursued.
Sect 7.0 – Describes the Quality Assurance provisions.
Sect 8.0 – Defines how documentation will be processed.

Volume 2:

Provides resource requirements, funding, special agreements, detail procedures, facility descriptions, detail schedules.

The Software Life Cycle Management Plan's greatest value is that it establishes consensus between the many system managers on what the policy, long term schedule, and group interface/responsibilities will be for a particular software system. This kind of long range planning performed at the commencement of a new software development is extremely supportive of the outyear maintenance effort occurring after the software becomes operational in the fleet. From the early planning data, high level management can acquire an appreciation of the many stages of activity and the estimated cost that will be required for the software development and maintenance support. Additionally, from this early planning, management can accomplish during the time of development of the software system the reservation of funding and the establishment of resources and software support facilities that will be required for the many years of maintenance on the software.

Other key planning which supplements the Software Life Cycle Management Plan and is required to be formulated during the early stages of the software system development is:

Configuration Management Plan
Quality Assurance Plan

Configuration Management planning is essential because the software will be under continual functional modification and improvement throughout its life cycle in order to overcome changing threat conditions. The continuous and numerous functional changes to the software will require stringent configuration control of the software baseline to avoid confusion and provide for orderly implementation of the software changes. For this purpose the required software configuration control is defined in the Configuration Management Plan.

Quality Assurance planning is likewise essential because of the critical use of tactical software in the defense posture of the Navy. The Quality Assurance Plan will define the test and evaluation methodology required to insure that the fleet receives software that has the highest functional effectiveness and reliability.

Software Management Organization. Definition of the software management organization for a software system during the life of the software system is important. There are many players that are required to provide engineering services during the life of the software. Therefore in order to minimize communication gaps and assign accountability for all the work that must be performed during the life cycle of the software, the management organization must be identified and responsibilities clearly assigned. In general, the software management organization for tactical software would be similar to the diagram in Figure 2.

Note that each tactical software system has its own Software Change Review Board to assist the Program Manager in exercising configuration control of the software. The membership of the Software Change Review Board consists of fleet users of the software plus representatives from each of the groups shown in Figure 2. The Software Change Review Board reviews proposed software changes in order to assess impact on fleet capability and advise the Program Manager on the level of priority that should be assigned to the changes. This is an important contribution that the Board makes because requests from fleet operators for software changes are generally numerous; however, available funding and personnel resources for implementing software changes are limited.

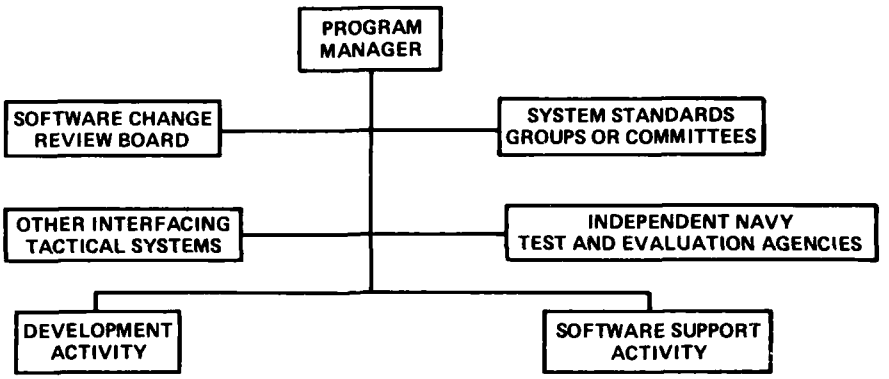


FIGURE 2 – Typical Software Management Organization

Design Factors. There are a number of software design factors that are significant to software maintenance that should be implemented in the initial development of the software. These factors are:

a. Employ top-down structured programming with standard language constructs when developing the software.

b. Use high level language.

c. Modularize the software architecture by breaking it up into subroutines restricted to one function each.

d. Define consistent software interfaces/protocol for calling or passing data between software subsystems.

e. Establish documentation standards and require that documentation be developed that will support the maintenance of the software.

f. Instrument the software by implementing test subroutines in line with the operational software code which will permit analysis of the utilization of computer resources such as use of transient memory, task execution time, interrupt processing, etc.

g. Provide a hardware breakpoint monitoring capability in order to halt execution of program to examine program faults.

h. Develop system data replay software to examine inner program data processing.

i. Establish a core and timing reserve during the development phase to enable future functional expansion of software during the maintenance phase.

There is nothing new or innovative in this list. Navy experience indicates however that incorporating this set of factors in the initial software design can simplify the task of making changes to the software system after the software becomes operational within the fleet.

Software Configuration Management. The Navy considers Configuration Management as a vital and extensive process that must be omni-present through the life of the software. This process is formal; that is, verbal change requests are not recognized by the process. Software change requests must be submitted in writing and the processing of the change request is controlled in a relatively formal procedure.

An important concept within Configuration Management is the identification of the software baseline. This concept can be understood through discussion of the following configuration management terminology:

1. Configuration Baseline
2. Configuration Item
3. Configuration Identification List

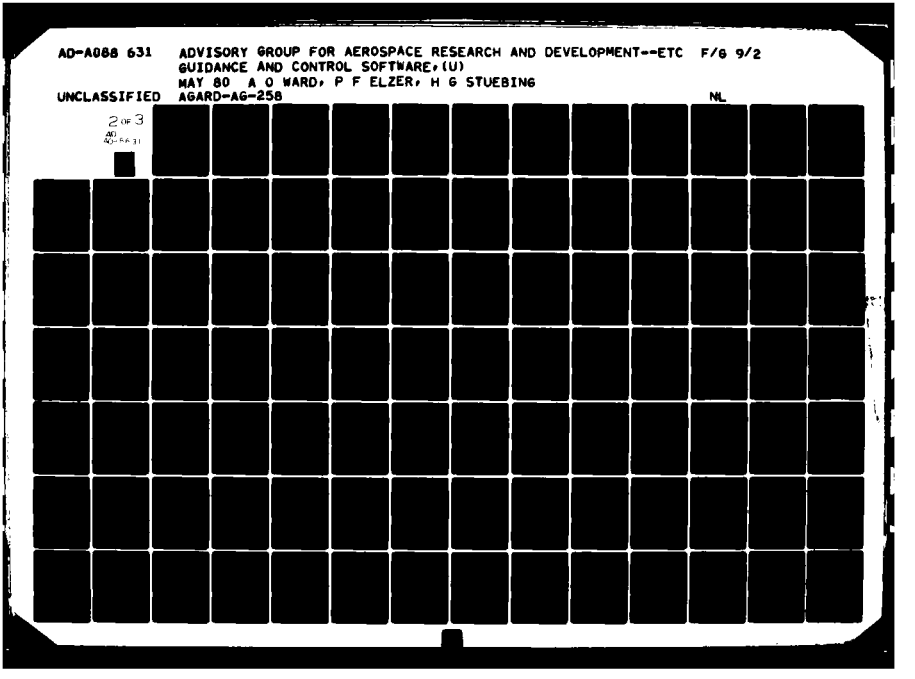
The Configuration Baseline is the actual software code before incorporation of a group of approved software changes. The Configuration Baseline is then separated into a number of Configuration Items which correspond one-to-one with the major subprograms or subsystems of the total software system. Each Configuration Item is then technically specified by a set of documents containing functional description, logic design, interface description, test plan/procedures, operator manual, etc. The group or list of documents that comprise the technical specification is known as the Configuration Identification List.

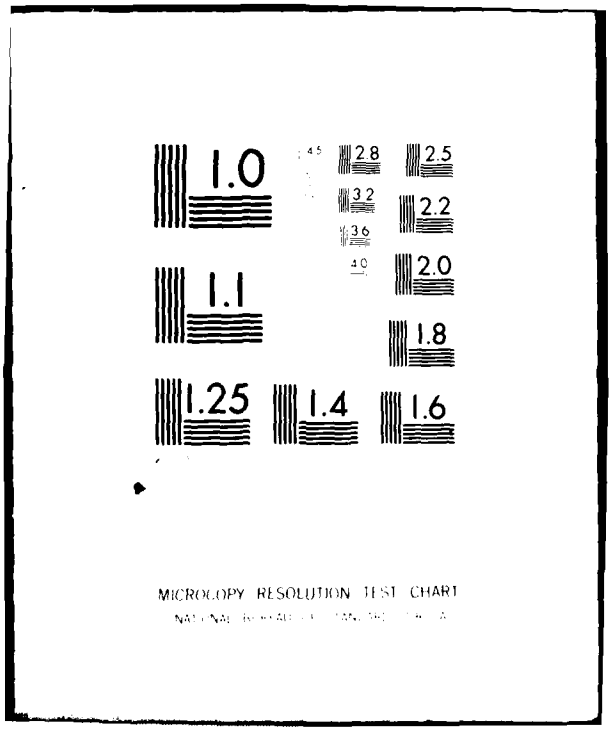
The control of software changes to the Configuration Items of the Configuration Baseline is the primary activity and essence of Configuration Management. The Configuration Identification List supports the control by defining the identity of the software Configuration Baseline. All changes requested by the software users must be analyzed in order to determine what software subsystems (i.e., Configuration Items) will be affected and what will be the cost for the implementation of the changes. These software change impact analyses are reviewed by the Software Change Review Board. If approved for implementation, the software code will be appropriately modified and the technical documentation of the affected Configuration Items will be changed in order that the documentation will accurately reflect the modified code. The resulting new version of the software code and the modified documentation constitute the next software Configuration Baseline and Configuration Identification List.

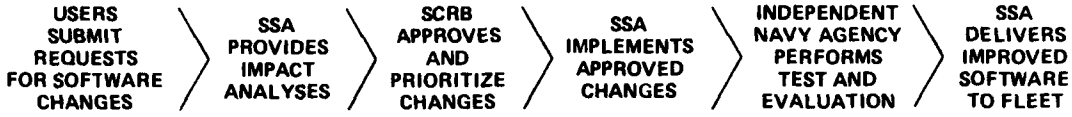
Configuration Management procedures are closely related to the steps that are taken to implement a software change. Figure 3 provides a general outline of the primary steps taken within the software maintenance process to incorporate a change and release improved software to the fleet.

AD-A088 631 ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 9/2
GUIDANCE AND CONTROL SOFTWARE,(U)
MAY 80 A O WARD, P F ELZER, H G STUEBING
UNCLASSIFIED AGARD-AG-258 NL

2 OF 3
AD A088631




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART



Note: SSA – Software Support Activity
SCRB – Software Change Review Board

FIGURE 3 – Simplified Software Maintenance Process

Utilizing the outline of Figure 3 as background for the ensuing discussion, this paper will now address how the control of software change is made possible by constraining changes to the following procedures:

a. The users of the software submit in writing on pre-designated forms, all requests for software corrections or new functional capabilities. Word of mouth or informal change requests are not recognized by the Configuration Management procedures.

b. The Software Change Review Board secretariat formally receives, logs, acknowledges to the users, and distributes the change requests to the Software Support Activity and members of the Software Change Review Board.

c. The Software Support Activity engineers technically determine and document how correction or new functional capability will change the baseline (i.e., which Configuration Item(s) will be changed) and how much will it cost to implement the changes.

d. The Software Change Review Board reviews the baseline change impact analyses and recommends to the Program Manager an implementation priority for each of the changes.

e. After receiving direction from the Program Manager on what changes to implement, the Software Support Activity implements the approved changes and concurrently tracks the progress achieved in establishing the next software Configuration Baseline.

f. Through operational testing the Independent Navy Test and Evaluation Agency will validate that the updated or new software baseline is in accordance with Program Manager approved changes and functional specifications.

g. Upon delivery of the new version of the software to the fleet, the Software Support Activity will apprise the fleet of the functional capabilities of the modified software baseline.

In order to facilitate Configuration Management, the Navy uses automated Configuration Management data bases. These data bases are resident on laboratory computer systems through the use of data base management system software programs especially designed to create a data base, update the data, and provide reports. The automated configuration management data base provides various formal and periodic Configuration Status Accounting reports such as:

Configuration Identification List
Software Change Request/Program Trouble Report Status
System Problem Report Status
Software Change Notice Reports

Additionally, these automated data bases are usually equipped with an "immediate access" feature which permits informal interrogation of the data base. Inquiries such as "List the change requests impacting the Navigation Subsystem with an implementation cost greater than $5000" can be easily and quickly made.

Software Quality Assurance Process. The Navy places serious emphasis on software Quality Assurance. This is demonstrated by considerable expenditure of funds in order to guarantee a high reliability for tactical software. The Quality Assurance process relies on two features:

1. The software design is verified early in the development of new or modified software. The rationale is that design problems are less costly to correct in the initial software development stages.

2. Quality Assurance is performed by a group of engineers that is independent of the group that is directly designing and coding the software modifications or new software. Since the Quality Assurance group has no personal investment in the design and development of the software, they do not suffer from a lack of objectivity. The independent Quality Assurance group performs its function with the firm belief that the software does contain errors that need to be determined and corrected.

Figure 4 provides a schematic outline of the software Quality Assurance process.

The Quality Assurance process begins with a number of software engineers or programmers laying-out design specifications for the development or modification of various software modules and/or subprograms. An independent Quality Assurance group provides early verification by analyzing the design specifications through examination of functional descriptions, performance specifications, flow charts, interface protocol, etc. and independently determines that the software design will satisfy the functional requirements. This early verification is completed before any code implementation.

After Quality Assurance approval of the design, coding commences and is followed by three levels of testing: subprogram tests, program tests, and program integration tests. These tests are performed by the software production group, but are observed by the Quality Assurance group. Upon completion of this testing, the coding effort is frozen and an interim program tape is delivered to the Quality Assurance group. The Quality Assurance group then performs independent laboratory functional and performance testing, core utilization tests and timing/loading tests in order to validate that the software end-product meets functional/performance requirements at the laboratory level.

After successful completion of the above Quality Assurance validation tests, the program development is frozen and an independent Navy Agency commences formal Test and Evaluation and validates that the software is operational in the fleet environment. Successful completion of formal Navy Test and Evaluation provides final Navy acceptance of the tactical software. This Quality Assurance process remains in effect during the life of the software.

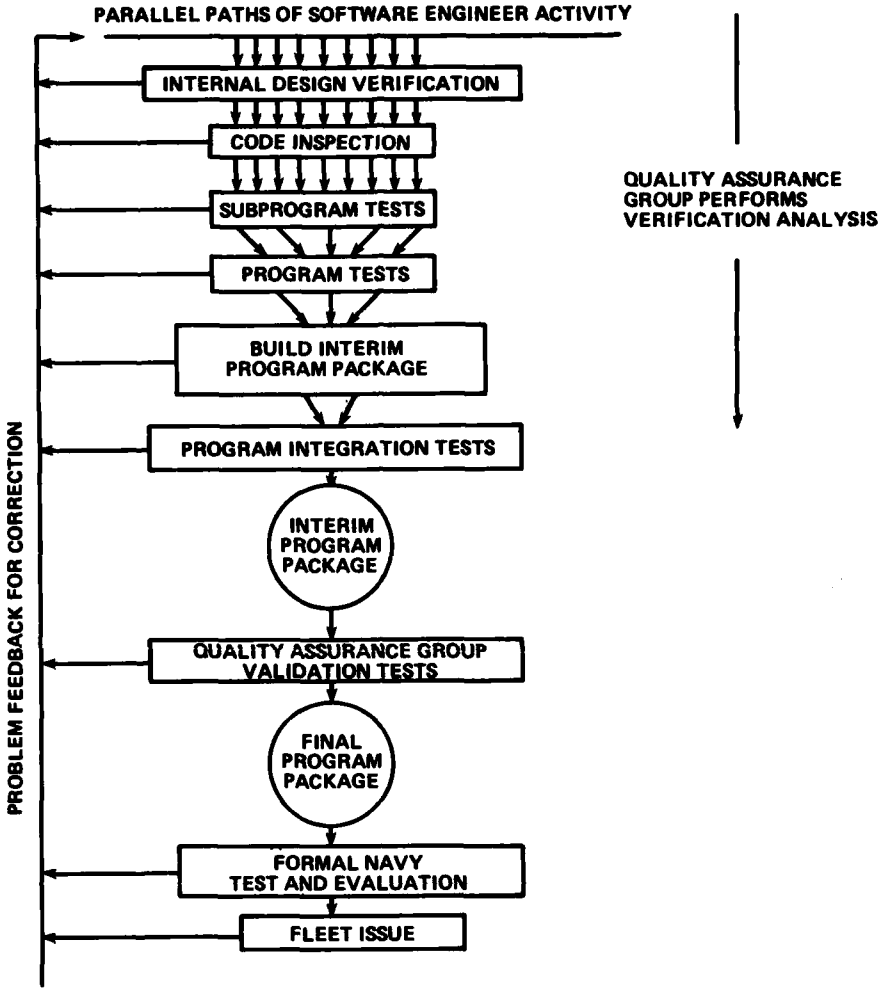


FIGURE 4 – Software Quality Assurance Process

Work Breakdown and Staffing. The Navy uses a Work Breakdown Structure to determine how the work is to be broken down into areas of effort. A Work Breakdown Structure is a management tool utilized in planning the progression of work, assigning responsibilities, and providing a framework for financial control and progress tracking. The major work breakdown categories for Phase 3 Navy Support (i.e., software maintenance) of tactical software are:

Engineering Management. Provide direction, plans, and schedules.
Configuration Management. Provide configuration baseline control.
Quality Assurance. Provide verification and validation.
Software Production. Provide engineering analysis, design, code and debug.
Laboratory Facilities. Provide operation and maintenance of software generation, integration, and training facilities.

These work breakdown categories illustrated in a Work Breakdown Structure format would appear as a hierarchical block diagram such as shown in Figure 5. Although not discussed in this paper, note the need to support the operations and equipment maintenance of Laboratory and Trainer Facilities. Because the fleet operating environment is not conducive to the production of software and specialized crew training, it is essential for the support of tactical software to provide for these facilities in order to conduct maintenance and training on the tactical software. Laboratory Facilities must be made available to the Software Support Activity in order to provide the ability to:

a. Generate machine loadable code.
b. Debug and test the modified software on a simulated tactical system.

Staffing for maintenance of tactical software can be divided into several working groups:

Program Office
Configuration Management
Quality Assurance
Software Production
Laboratory Operations

{ Tactical Mission Software Group
System Test Software Group
Support Software Group
Trainer Software Group

The working groups correspond approximately to the Work Breakdown Structure. The Program Office is the management focal point for all the groups. Note that for Software Production there is a team for each of the software product lines which has the responsibility for the analysis, design, code, debug, and integration for its assigned software product. The precise organization for software maintenance of tactical software varies in accordance with the formal organization of particular Navy Agencies and Laboratories. The personnel resources for the working groups may be provided from agencies organized functionally or as a matrix. What is important, however, is that resources at least are assigned to informal working groups as shown above.

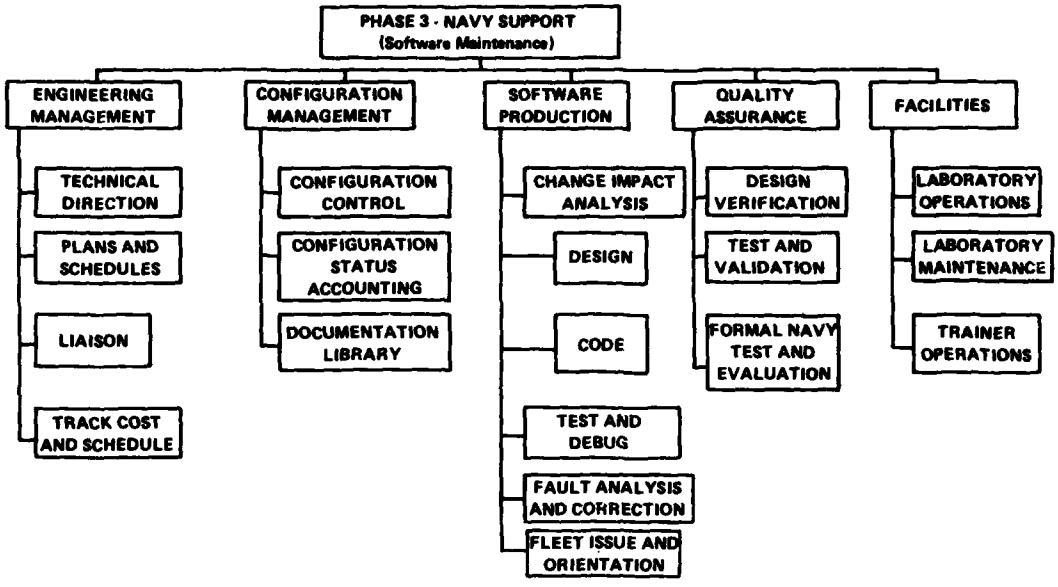


FIGURE 5 – Simplified Work Breakdown Structure

Software Maintenance Cost. The maintenance cost of software is very much related to the cost of developing the software. A rough rule-of-thumb estimate would be that the cost of software maintenance can be expected to be approximately equal to the entire cost of developing the software to the stage where it can be released for operational use. Some typical costs in terms of manpower *experienced for tactical software range between 800-2400 manyears required for the total life cycle support of tactical software. These* total manyears can be further separated into the three phases of the life cycle as follows:

	Phase 1 Development	Phase 2 Transition	Phase 3 Navy Support*
Required Manyears/Year	60-100	120-240	40-80
Duration of Phase	3-5 years	2-3 years	10-15 years
Percent of Life Cost	25%	25%	50%

* Phase 3-Navy Support is the software maintenance period inthe life of the software.

Note that hardware costs are not included and would have to be added in order to determine total system life cost. Assuming that the hardware requirements are available or provided for under a separate budget, the software cost is primarily for manpower needed to develop and maintain the software. Therefore manyear estimating is an excellent indicator for software cost.

The 50 percent of life cost shown for the software maintenance phase indicates that the maintenance of software is a significant contributor to the life cost of software. Obviously, management must be concerned and must plan early in Phase 1 - Development in order to:

a. Insure that the design of the software will result in a system that is relatively easy to maintain.
b. Reserve funding and establish personnel resources for the outyears of required maintenance.

Although software maintenance appears expensive, the important point for the reader to derive is that the Navy is acquiring new functional capability without having to scrap the old software and commence a new software development. This approach is the most cost effective if the software system is designed initially to accommodate maintenance, for example, initially design the software with a reserve for memory and timing to permit future expansion.

To determine the cost of software maintenance, an experienced software planner or manager develops a Work Breakdown Structure as a basis for his cost estimates. Figure 5 shown earlier in this paper is an example, but the software planner generally develops a work breakdown with more subdetail. When the Work Breakdown Structure is completed each of the work activities or blocks given in the structure are then laid out in a milestone or schedule chart as shown in Figure 6. The schedule shown is for an 18 month period which is generally the time required to develop an improved version of tactical software and re-issue the software to the fleet. The planner then assigns manning requirements for each work activity. The sum of all manning levels given for the work activities constitute the total manpower resources required. Typical manloading is shown in Figure 6 by the parenthesized numbers and the manloading sum at the bottom of the figure.

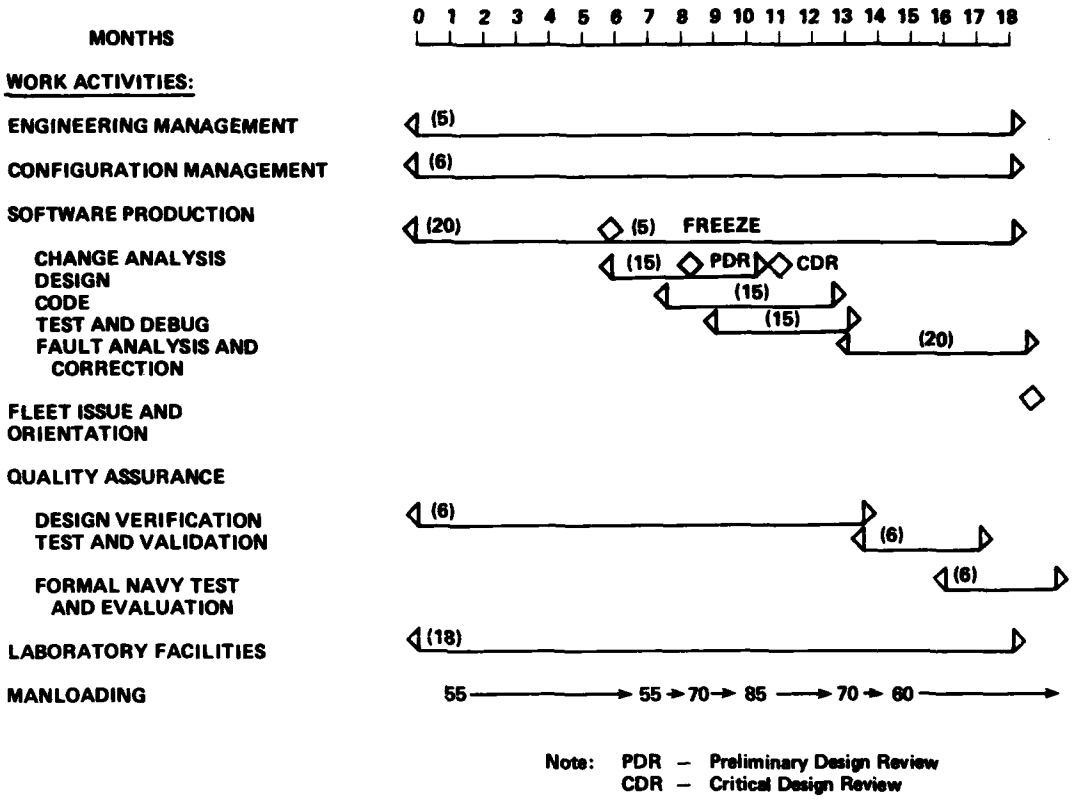


FIGURE 6 – Simplified Navy Support Phase Schedule With Manloading

The manning estimates developed by the software planner are dependent upon a number of factors:

a. size and complexity of the software
b. anticipated number of requests for software change
c. urgency to implement changes

The urgency to implement software changes will be dependent upon the use of the software. If the software is used for national defense, the urgency can be expected to be high. The number of requests for change will generally be directly proportional to the number of users of the software. The size and complexity of the software will affect the ease or difficulty in which the software changes can be implemented. Complexity of the software increases by an order of magnitude when the system, for example, is a real time application, has numerous and diverse functions to perform, provides for multiple man-machine interfaces, and controls the operation of a variety of hardware external to the central computer.

The following outlines the steps that are taken to determine approximate estimates of Phase 3 maintenance cost:

1. Divide software into major subsystems, determine sizes and languages and assess complexity of functions and interfaces.
2. Estimate average number of requests per year and urgency for software change.
3. Develop a Work Breakdown Structure for each major software subsystem and schedule work activities over a fleet re-issue cycle.
4. Determine manning requirement for each scheduled work activity.
5. Convert staffing estimates to cost.
6. Add in nominal estimates for any equipment rental and material costs.
7. Sum items 5 and 6 to determine total re-issue cycle cost.
8. Estimate number of Phase 3 life re-issue cycles to determine total cost.
9. Adjust cycle cost by an inflation factor.
10. Decrease or taildown manning resources for last few cycles to provide manpower for next new software development.
11. Check total Phase 3 maintenance cost by comparing to estimated sum of Development Phase 1 and Transition Phase 2 cost. Phase 3 should be approximately equal to the sum of Phase 1 and 2.

The above outline is a moderately simple approach to determine maintenance cost. Yearly budgets in actual use will fluctuate depending upon the urgency for software change in a given year. These fluctuations however are difficult to forecast. The best approach has been to determine manning levels essentially based on software complexity, number of users, predetermined Work Breakdown Structure of activities, and a predicted schedule for the development of software re-issues.

CONCLUSION

The foregoing discussion has reviewed the salient concepts that Navy management addresses in providing for the maintenance of tactical software. Because of the criticality to defense readiness and the multimillion dollar cost of fleet software logistics, the software maintenance management philosophy is under continual scrutiny by Navy management and as a consequence, Navy policies will change as new methodologies and concepts are developed by the U.S. Government and private industry.

The most important features of Navy management's approach to software maintenance are:

1. Software maintenance is a technically complex task.
2. Software maintenance requires a strong and well defined Configuration Management and Quality Assurance.
3. Software maintenance must have a set of documentation which as a minimum should include a definition of functional requirements, design specification, operator guide, and Software Life Cycle Management Plan.
4. Software maintenance requires laboratory facilities and support software adequately designed and available.
5. Software maintenance provides timely functional improvements to software that will overcome changing threat conditions.
6. Software maintenance is expensive and therefore must be considered and thoroughly planned for early in the development of a software system.
7. Software maintenance is cost effective in the long run because it provides re-issues of functionally enhanced software to the fleet over a long period of time. The periodic re-issue of functionally enhanced software appreciably extends the time before costly major software redesign and development of a new software system must be undertaken in order to replace obsolete software.

DATA SYSTEM FOR THE INFRA-RED ASTRONOMICAL SATELLITE (IRAS)

by
R.C. van Holtz
IRAS Mission Operations Manager
National Aerospace Laboratory NLR
Anthony Fokkerweg 2
1059 CM Amsterdam
the Netherlands

SUMMARY

This chapter discusses the data system that is being developed for the Infra-Red Astronomical Satellite (IRAS). After introducing the satellite and its objectives, the setup of the on board and the ground data system is described. The subsystem responsible for the control of the satellite, called IRAS Ground Operations, is then exposed in more detail. In particular, attention is given to the observation planning system envisaged for this satellite.

Finally, the configuration control mechanism for the data system is portrayed, since it is felt to be rather exemplary for such a complex international project.

CONTENTS

ABBREVIATIONS

ACTADM	Account Administration program
ADD	Ancillary Data Distribution assembly
AND	Alpha-Numeric Display
ANS	Astronomical Netherlands Satellite
ARC	Ames Research Center
BASD	Ball Aerospace Systems Division
CPC	Chopped Photometric Channel
DAX	Dutch Additional Experiment
DDPS	Digital Data Processing System
DISCOV	Survey-Coverage Display program
DODR	Digital Original Data Record
DRA	Digitizing and Reduction Assembly
EGA	ETL Generation Assembly

ERD	Event Related Data
ETL	Experiment Target List
ETLARC	ETL "Archiving" program
GANS	Generation Aid for Non-Survey observations
GAS	Generation Aid for Survey observations
GSFC	Goddard Space Flight Center
HST	High Speed Telemetry (HS-TLM)
ICD	Interface Control Document
ICIRAS	Industrial Consortium IRAS
IGO	IRAS Ground Operations
IR	Infra-Red
IRAS	Infra-Red Astronomical Satellite
JISWG	Joint IRAS Science Working Group
JPRD	Joint Project Requirements Document
LOFOP	Print program for the "long-term forecast"
LRS	Low Resolution Spectrometer
LST	Low Speed Telemetry (LS-TLM)
NASA	National Aeronautics and Space Administration
NASCOM	NASA Communications Network
NIVR	Netherlands Agency for Aerospace Programs
NL	the Netherlands
NLR	Dutch National Aerospace Laboratory
NOM	Non-survey Observation Module
NONSANS	Non-Survey Analysis System
NSSDC	National Space Science Data Center
OAM	Observation Administration Module
OBS	On Board Software
OBSADM	Observation Administration program
OCI	OBS Change Implementation assembly
OCC	Operations Control Centre
ODA	Orbit Determination Assembly
OHA	Operational Hardware Assembly
OPA	Operational Procedures Assembly
ORT	Operations Real Time assembly
OSM	Observation Scheduling Module
PAF	Preliminary Analysis Facility
PSG	Pass Schedule Generation assembly
PRADM	Print program of the OAM
RAM	Random Access Memory
REGEN	ETL Generation program of OSM
RFS	Radio Frequency Subsystem
ROG	Space research group of the Groningen University
ROM	Read Only Memory
SDAS	Scientific Data Analysis System
SDS	Special Data Storage
SEA	Satellite Evaluation Assembly
SHK	Stored Housekeeping data
SHOFOP	Print program for the "short term forecast"
SKB	Standard Keyboard
SOG	SOP Generation assembly
SOM	Survey Observation Module
SOP	Satellite Observation Program
SRC	Science Research Council
SRM	Survey Recovery Module
STA	Software Testtool Assembly
STDN	Spacecraft Tracking and Data Network
SURE	Survey Recovery program
SWC	Short Wavelength Channel
TSY	Telescope system
UK	United Kingdom
UPMAF	Program to maintain in NOM
UPSUP	Program to update survey parameters
US	United States of America
UTA	Utility Assembly
UTC	Universal Time Co-ordinated

1. INTRODUCTION

The objectives of IRAS are to execute an astronomical all sky survey in the infra-red, and to perform additional observations of selected sources and regions to allow a more detailed analysis. Such a mission has not been carried out before, which accounts for the, sometimes large, uncertainties in the requirements placed on the data system.

IRAS is due to be launched in the fall of 1981, in a near-polar, circular orbit with an altitude of 900 km. The inclination aimed for is 99° which will yield a precession rate of 360°/year, making it sunsynchronous. In addition, the injection parameters are chosen such that the satellite will come into a twilight orbit. This gives eclipse-free periods of approx. 10 months every year. The orbit selected makes it in principle possible to observe any part of the sky within 6 months (Fig. 1).

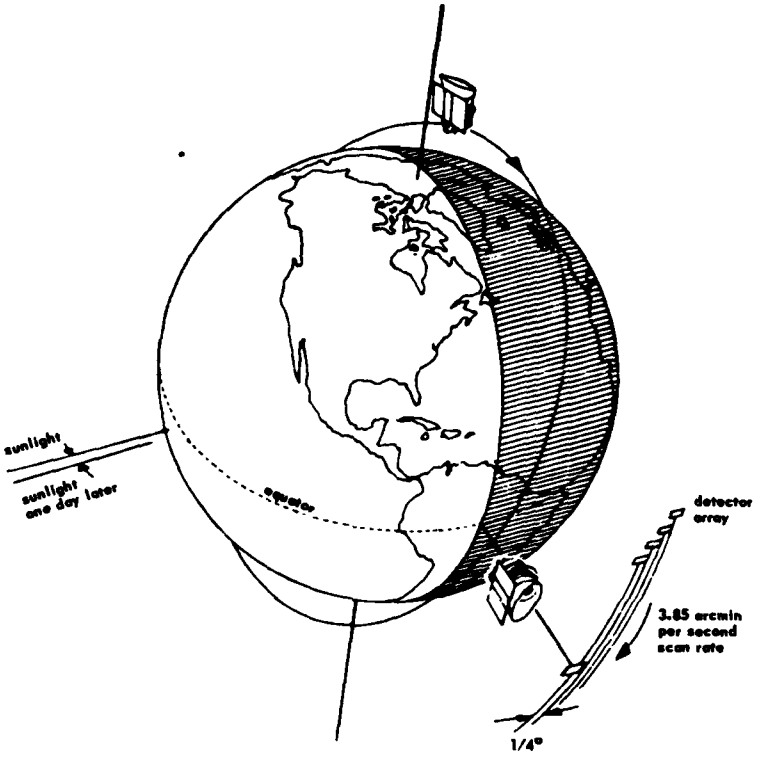


Fig. 1 Principle of sky survey

The observations will be performed through a 2-mirror 60 cm telescope giving a 63 arcmins unvignetted field of view. To achieve the goals set by the astronomers to the survey-sensitivity, it proved to be necessary to reduce the radiation from the telescope itself by cooling it to ≈ 10 K. This is accomplished by mounting the telescope and its focal plane assembly in a dewar containing, initially, 76 kg of superfluid helium. This amount of helium will allow an operational lifetime of one year, provided a number of precautions are taken with regard to the pointing direction of the telescope. Pointing it to the sun will result in a, practically, immediate loss of the mission, while pointing towards the earth will reduce the operational lifetime to something in the order of 30 minutes. It is the task of the attitude control system to prevent the occurrance of such unfortunate events, once the protective cover has been ejected from the telescope. A cutaway view of the telescope is presented in figure 2.

The focal plane of the telescope houses the detectors of the two scientific instruments and of the starsensor used in the attitude control system (Fig. 3). The scientific instruments are:

- the survey array of 62 detectors, divided over four wavelength bands from 8 to 120 μm, with a swath-width of 30 arcmins; the datarate coming from this instrument, after encoding by an internal microprocessor, is 6272 bps
- the so-called Dutch Additional Experiment (DAX) which is divided into three parts:
  - a short wavelength channel (SWC), providing statistical data on the number of IR-sources in the 5-8 μm band; its normal data rate is mere 8 bps but it can be commanded to give an additional 512 bps
  - a low resolution spectrometer (LRS), to aid in the classification of sources detected by the survey array; it gives a 5 % resolution between 6 and 24 μm; the datarate is 1280 bps
  - a chopped photometric channel (CPC) for mapping infra-red sources in the 45-120 μm band; its datarate is 1562 bps; the CPC cannot be used simultaneously with the other instruments.

Since there are 14 orbits per day and the precession rate of the orbit is almost 1 °/day, the swath-width of 30 arcmins of the survey array would allow a repetition rate in coverage of seven at the ecliptic. The astronomers have opted for a repetition rate of 6 but with specific constraints on the intervals between coverage. This to allow the processing to identify moving objects such as asteroids, whilst minimising the risk of missing actual sources. Coverage of the areas near the ecliptic poles will in principal be restricted to 6 as well. It is expected that roughly 60 % of the one year lifetime of the

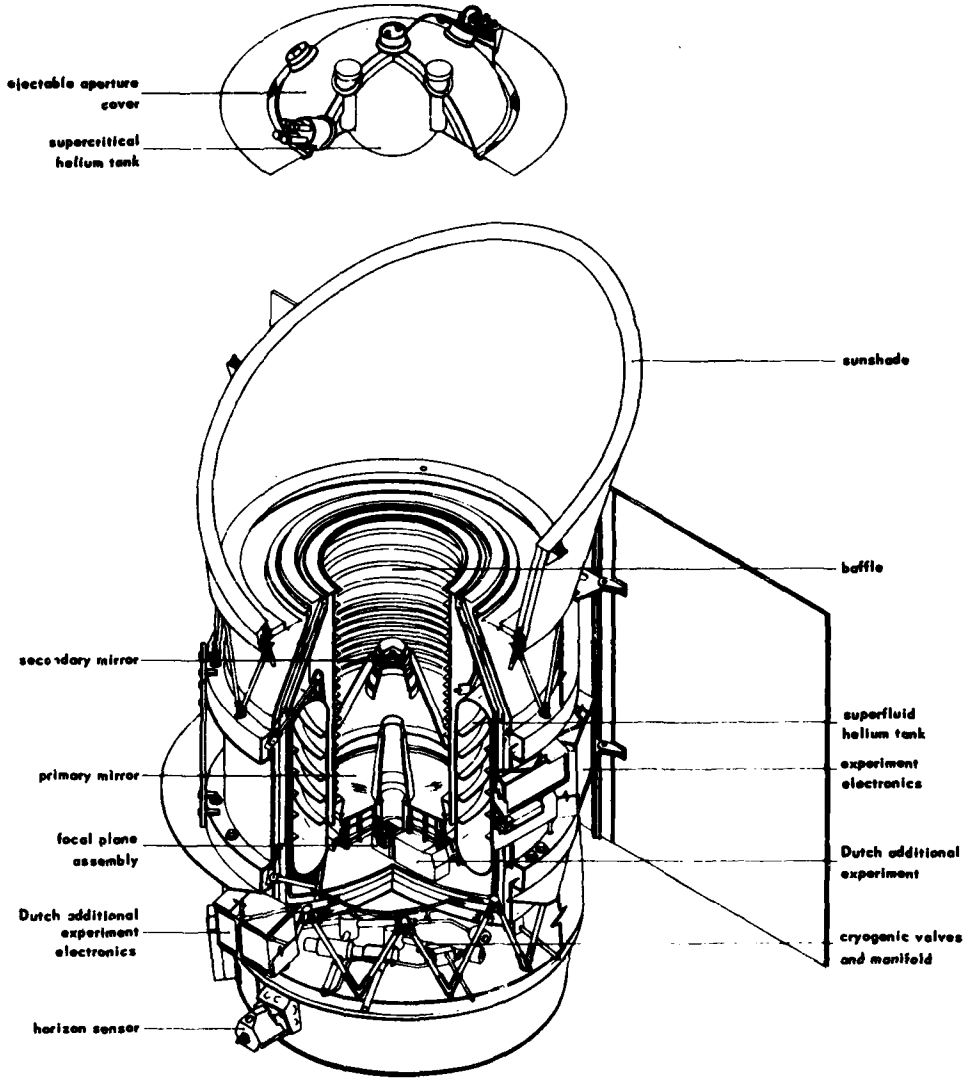


Fig. 2 Cut-away view of the telescope

satellite will be spent on the survey, the remainder of the time will be used for performing the necessary instrument and spacecraft calibration sequences and for the additional observations.

2. PROJECT STRUCTURE

Before going into more detail, it is necessary to briefly explain the project structure. Otherwise the specific setup of the data system may be rather difficult to apprehend. IRAS is a collaborative project involving the United States, the United Kingdom and the Netherlands. It is managed by the Joint Project Executive Group, which is composed of members of the different organisations participating. The leading organisation of the various countries for this project are:

- National Aeronautics and Space Administrations (NASA) Headquarters
- Science Research Council (SRC)
- Netherlands Agency for Aerospace Programs (NIVR).

The scientific requirements for the mission are set by the group of astronomers appointed by these leading organisations: the Joint Infra-Red Science Working Group (JISWG). The Project Executive Group is further assisted by a number of joint working groups, composed of experts of the institutes responsible for implementing the work. For the present description the Joint Data System Working Group is the most relevant one to mention. Approximately three times a year the joint groups meet. Apart from the normal technical interface discussions, these meetings are also used for reviewing the various aspects and elements of the ground data system.

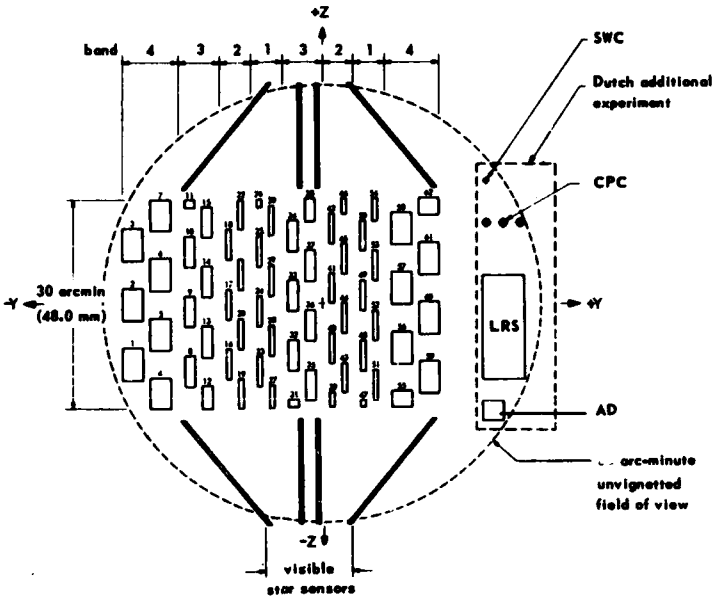


Fig. 3 Focal plane layout

The task division over the three countries is as follows:

- US: . the construction of the telescope and the survey array instrument
  . satellite launch (2-stage Delta 3910 configuration with 9 strap-on boosters)
  . final science data processing for the survey array instrument
  . provision of some standard satellite items (NASA standard transponder, recorder)
- UK: . the provision of the project Operations Control Centre
  . provision of the prime tracking station
  . preliminary science analysis of the data of the survey instruments
  . write part of the software needed for the operations
- NL: . the construction of the spacecraft
  . construction of the three instruments in the DAX
  . final science data processing for these instruments
  . the integration of the overall satellite
  . design and development of the system to operate the satellite; this includes the on board software system

In the US the project management has been allocated to the Jet Propulsion Laboratory (JPL). This laboratory also provides the final science data processing. The telescope management is performed by Ames Research Centre (ARC), with Ball Aerospace Systems Division (BASD) as the prime contractor. Launcher management is carried out by Goddard Space Flight Centre (GSFC).

In the UK the project management is placed with the Appleton Laboratory, which also carries out the great majority of the work. The Operations Control Centre is built at the Chilton site of the Science Research Council. The prime tracking station is located there as well. It is built around a 12 meter dish, temporarily on loan from NASA.

In the Netherlands the project management is performed by an industrial consortium, ICIRAS, with the exception of the Dutch Additional Experiment. ICIRAS consists of Fokker and Signaal, with as one of its larger subcontractors the National Aerospace Laboratory (NLR). The DAX is built by the space research

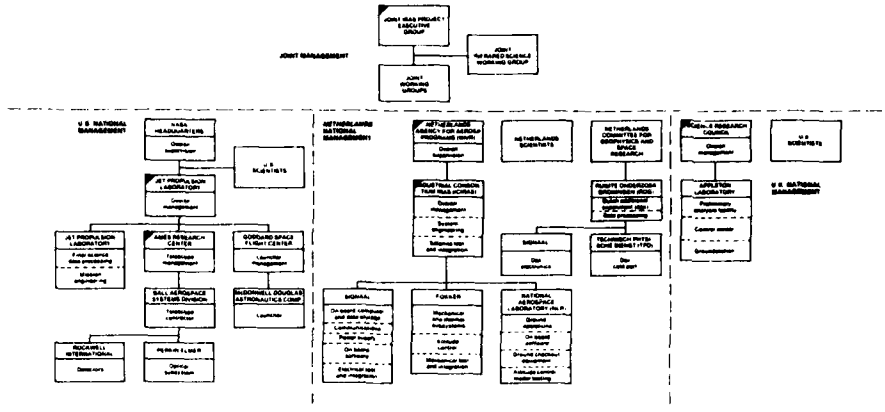


Fig. 4 IRAS organisation/management structure

group of the University of Groningen (ROG), which is also responsible for the final science data processing for these instruments.

A more complete diagram of the organisation and management structure for IRAS is shown in figure 4.

3. SATELLITE DESCRIPTION

The dewar containing the telescope and the instruments is mounted on the spacecraft platform, which houses the majority of the units of the other satellite subsystems, such as (Fig. 5)

- power control subsystem, consisting of a battery and electronics to regulate, convert and distribute the power obtained from the solar panels
- attitude control system, using fine and coarse sunsensors, gyros, starsensor (detectors are in the focal plane assembly), horizon sensor and magnetometer as sensors, and reaction wheels as actuators of which the momentum can be dumped through magnetic coils
- on board data system, consisting of two redundant computers and taperecorders, for control of the satellite and data collection/storage

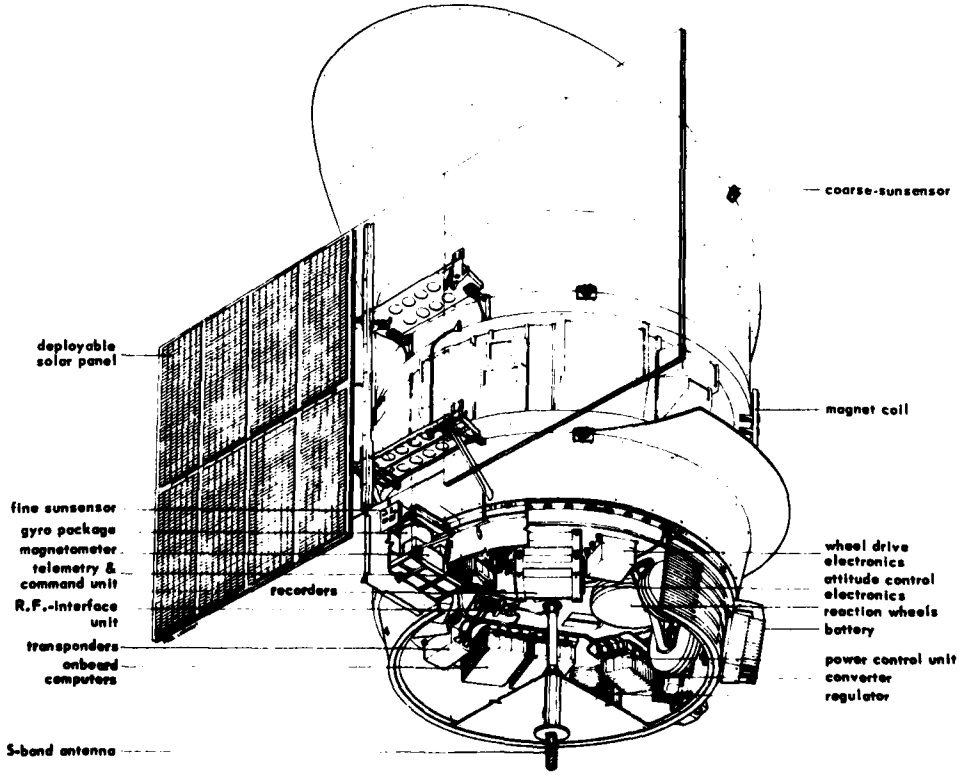


Fig. 5 Spacecraft unit layout

- radio-frequency subsystem, consisting of an S-band antenna, diplexerhybrid, two redundant transponders (NASA standard) and a telemetry and command unit.

The total dimensions of IRAS are 3.6x3.2x2.1 m and its weight at launch is 1020 kg.

The mission requires a 3-axis stabilized satellite, with an attitude control accuracy better than 5 arcmin and a limitcycle of less than 10 arcsec. Two basic attitude modes are provided for the observations:

- scan mode, in which the scan velocity can be varied: for the survey the velocity is set at 110 % of the orbital rate (3.85^{o}/min), a velocity of zero is effectively a pointing mode
- raster scan mode, which consists of a number of parallel scans to cover extended objects in one observation; variables are the scan speed, the stepsize and the scan-length; when the stepsize is set to zero, a number of small scans will be made over the same sky-area, which effectively can be used to increase the sensitivity, after processing (so-called deep-sky-survey).

In performing the various manoeuvres care must be taken that the attitude remains such that no undue heat input from the sun or the moon occurs (helium boil-off), that the performance of the telescope is not excessively degraded (dust gathered on the mirror-surfaces), and that the instruments do not get saturated by bright objects in the sky, like Saturn and Jupiter (temporary "blindness"). It is clear that this variety of constraints severely limits the freedom in pointing of the telescope boresight. The logic of the attitude control system is designed to guard against attitudes which would cause a fast helium boil-off. The other constraints are "softer" and have to be taken into account in the observation planning facility of the ground data system. No on board protection is envisaged for these constraints.

The operations will in principle be controlled from a single tracking station, located in the UK near Chilton, Oxfordshire. The orbit is such that the satellite will be visible on approx. 3 successive orbits around dawn and 3 around dusk. For various reasons the operations are designed on the basis of the use of only one of these passes every morning and evening, introducing a "natural operational cycle" of 12 hours average. During each of these real-time contacts with the tracking station, the satellite is provided with a time-tagged list of observations and manoeuvres it has to perform. This is known as the Satellite Observation Program (SOP). The satellite must be able to store and execute such SOPs, and to store the data gathered in these periods. The on-board data system therefore comprises a re-programmable, redundant, computersystem and two taperecorders. Since the satellite will be out of sight (and control) of the control centre for extended periods (up to a couple of days in case of problems on the ground) the overall satellite design must have a high immunity against minor anomalies. This specifically applies to the attitude control system as errors in attitude can have dramatic effects.

4. ON BOARD DATA SYSTEM

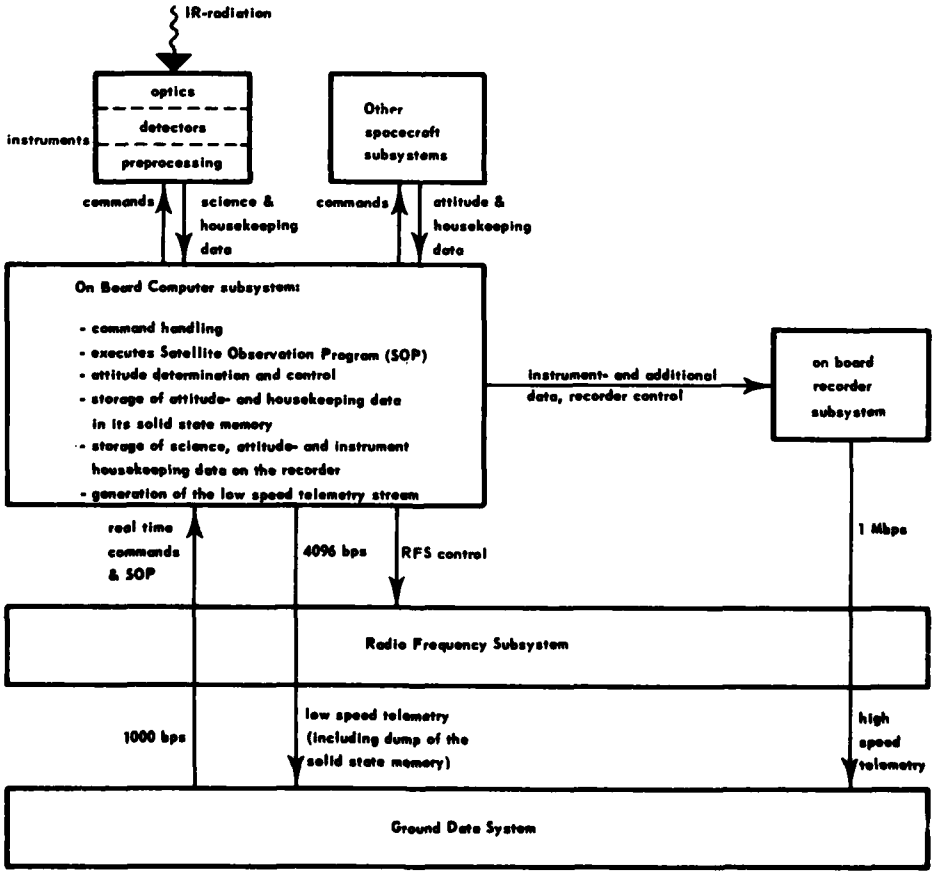


Fig. 6 IRAS on-board data system

The on board data system has two main functions. Firstly, it has to transfer the science information from the detectors to the ground. With science information is meant both actual detector output data and the housekeeping type of information necessary for the processing of the detector data (temperatures, satellite attitude, etc.). Secondly, the data system is used for the control of the satellite, in its broadest sense: commanding (both real-time and time-tagged, through the Satellite Observation Program), attitude control, on board health monitoring (e.g. battery temperature) and to provide the data for health monitoring and performance evaluation by the ground data system. Figure 6 depicts the setup of the on board data system for these functions.

The science data generated by the detectors is encoded by pre-processor units built in the instruments. For the survey array the pre-processing is performed by a redundant, 16 bits, micro-computer system. For the instruments of the DAX it is done by "standard" electronics. The pre-processing units also collect and "format" the instrument housekeeping data needed for control and data processing. The data from the instruments are collected by the on board computer at that moment, are augmented with the necessary attitude information and "formatted" for storage by one of the two taperecorders. A number of storage formats can be selected, via the SOP, to suit the different observations. The size of the formats is fixed at 1024 bytes . The recorder data is played back through the Radio Frequency Subsystem (RFS) once every 12-hours roughly, during a pass over the prime station. The average amount of data stored during such a period is 350Mbits. The maximum is determined by the capacity of the taperecorders, which is 455 Mbits each. In order to be able to playback such amounts of data comfortably within the duration of a pass (10 to 15 minutes) a channel of 1 Mbps is used.

Not all the satellite data necessary for processing of the science data is stored by the taperecorders. This is done for reasons of efficient storage, as a number of these data are generated in frequently. For example, the attitude of the satellite is controlled on the basis of a gyro, for one axis. Errors in attitude due to the behaviour of the gyro can be corrected for, using the crossing times of known visual stars over the starsensor splits in the focal plane assembly. The number of stars that are sufficiently well known in position and are bright enough to guarantee recognition by the starsensor is such that on average a few starcrossings are expected per scan. The amount of data generated per starcrossing is relatively large compared to the size of the formats stored by the taperecorder. On the other hand, the total amount of data thus generated per observation is insignificant with respect to the total science data. These infrequently generated sets of information are therefore stored in the solid state memory, together with all the housekeeping data necessary for control of the satellite. The solid state memory data are transmitted to the ground via the low speed (4096 bps) beacon signal, which also provides the information for real-time satellite control.

As mentioned earlier, commanding of the satellite can be done either in real-time or time-tagged through the SOP. In both cases, the on board computer distributes the commands to the units via the proper decoders. The only exception is for those commands that effect the status of the on board computers themselves. They are decoded by special circuitry in the telemetry- and command-unit. The real-time commands and the memory-load messages for the on board computer (like the SOP) are uplinked via a 1000 bps channel.

The processor in the two on board computers is based on the Philips P850 mini-computer series. Both computers are equipped with 32 kwords (16-bit) of memory divided over two blocks. For redundancy reasons, each processor can be connected with any two of the four memory blocks of the total system. The communication with the other satellite units goes via a standard Computer Interface-bus (serial). Every unit is connected to both the SCI-buses. The power control unit informs the units which processor is currently active.

On the one hand the on board computer system should exhibit a great amount of flexibility, so that changing requirements can easily be implemented. Such changes can be induced by an incorrect launch, a better understanding of the behaviour of the satellite and especially the instruments, or by on board anomalies. On the other hand the on board computer has the task to ensure that the attitude of the satellite does not give rise to an untimely boil-off of the available helium. The flexibility is ensured by using part of the 32 kwords Random Access Memory for program storage. This in principle allows the in-flight modification of any program used on board, whether it be for datastorage or even for attitude control. The program in the RAM is rather vulnerable, as it can be distorted by software errors or by hardware errors, such as power-dips. The use of only a RAM based program would give an unacceptable high risk for the safety of the satellite. To eliminate this problem, each processor can be switched to a program, carried in a Read Only Memory of 3 kwords. This ROM program uses the most reliable attitude sensors to ensure a safe attitude. It further provides telemetry containing the basic housekeeping, parameters, it decodes and distributes most switching commands and, finally, allows the loading of programs into the RAM-area.

As the ROM space is rather limited, no provision is made in ROM-mode for executing SOPs nor for the sophisticated attitude control and data storage modes, normally required for performing observations. The prime objective of the ROM-mode is to keep the satellite in a safe attitude mode, long enough for the ground operations to react and correct, if possible, the anomaly. Further safety measures have been included in the on board data system and overall satellite design to ensure a timely detection of a potentially dangerous attitude of the satellite, even due to a failure of the processor in use, or of an attitude sensor.

The design of the on board computer is such that the ROM program always occupies the first 3 k of addresses, leaving 29 kwords of addresses for the RAM. In addition the ROM program requires a work area of about 0,5 k of RAM. The RAM-program itself is about 11 kwords long, including the space required for parameters and variables. The SOP area reserved is 4 kwords. The remainder is available for storage of satellite data, as required by the ground operations. More will be said on this during the discussion of the ground data system.

5. GROUND DATA SYSTEM REQUIREMENTS

The ground data system has to fulfil the following basic requirements:

. control the satellite, which covers the aspects of
  - orbit determination
  - generation of Satellite Observation Programs
  - commanding
  - satellite health assessment
  - satellite performance evaluation and maintenance
. make the science data available to IR-astronomers for analysis, so they can:
  - verify the accomplishment of the survey programme
  - verify the accomplishment of the additional observation programmes
  - derive follow-up observations, either for IRAS itself, or for any other satellite or observatory
. reduce the volume of the data to products which are manageable for individual astronomers for further analysis
. file all the data, from which these products are derived, in a retrievable form.

The main interfaces of the ground data system are with the satellite and with the IR-astronomers. Two information streams can be identified:

- observations defined by the astronomers, resulting in Satellite Observation Programs for the "12-hours" operational periods
- data generated by the satellite, processed and augmented with ground-generated additional information for presentation to the astronomers, for verification of the progress of the mission (short term products) and for further scientific analysis (long term products).

The other important interface of the ground data system is with the satellite engineers. During routine phase their support is rather restricted (unless satellite emergencies turn up) and is mainly of a monitoring nature. During the launch and in-orbit-checkout phases of the mission, the involvement of the engineers is much greater. In order to verify the performance of the instruments and of the spacecraft (especially the attitude control system) a large number of non-routine operations have to be carried out. Some of these operations will require temporary modifications of the on board software, but practically all of them will put additional processing requirements on the ground data system. Unfortunately it is common practice that these non-routine requirements are not identified until a later stage of the development programme, which is generally not before the overall satellite integration and test-phase. For IRAS it was attempted to identify these requirements earlier, so that they could be properly integrated into the design of the ground data system. Where it turned out to be difficult to timely identify these requirements, a number of "standard" specialities were included, based on experience with previous satellites.

It is worthwile to point out at this stage that the operators, who are controlling the satellite, the control centre, the tracking station and the data processing systems, are not considered to be external users of the IRAS ground data system, contrary to possible other practices. The same holds for the mission planners provided by the project.

6. GROUND DATA SYSTEM SETUP

The ground data system is composed of four major elements:

. IRAS Ground Operation (IGO), developed by the Dutch National Aerospace Laboratory and the Appleton Laboratory of the UK Science Research Council
. Preliminary Analysis Facility (PAF), developed by the Appleton Laboratory
. Scientific Data Analysis System (SDAS), developed by the Jet Propulsion Laboratory, in the US
. ROG Non-Survey Analysis System (ROG-NONSANS), developed by the space research group of the University of Groningen, in the Netherlands.

The IGO- and PAF-systems will be installed at the project Operations Control Centre (OCC), located at the Chilton site of the Science Research Council, together with the prime tracking station. The Jet Propulsion Laboratory in Pasadena, California, will house SDAS. It is currently planned that the system developed by ROG will be installed at the OCC until the end of the operations, when it will be moved back to the University of Groningen.

In addition to these project peculiar elements, use is made of the services provided by a number of NASA facilities:

- Spaceflight Tracking and Data Network (STDN), for early-orbit and emergency support
- NASA Communications Network (NASCOM), for the daily information transfer between the US and Europe, and for operating the STDN stations from the project Operations Control Centre
- National Space Science Data Centre (NSSDC) at Goddard Spaceflight Centre, for the ultimate filing of the products delivered to the science community.

6.1 Science information flow

The science information flow through the ground data system is depicted in figure 7. Only the four major elements have been indicated for clarity. Starting in the top left-hand corner, every morning and evening, IRAS Ground Operations provides the satellite with a new Satellite Observation Program and collects the data gathered in the past "12-hour" SOP-period. This is done either through IGO's own prime tracking station or, in case of anomalies, through an STDN tracking station, using 7.2 kbps NASCOM high speed data links. The SOP is a collection of planned observations for both the survey and non-survey programs, as defined by the IR-astronomers associated with IRAS. Below, more will be said on the tools that IGO will provide to facilitate this task. The science data, relayed through the high speed telemetry link of 1 Mbps, is digitised immediately after the pass. Some preprocessing is then applied for the Preliminary Analysis Facility and the ROG Non-Survey Analysis System. The on board solid state memory data coming through the low speed telemetry link is processed in parallel and the necessary housekeeping and attitude-related information is extracted. This, together with other ground derived information, such as orbital and

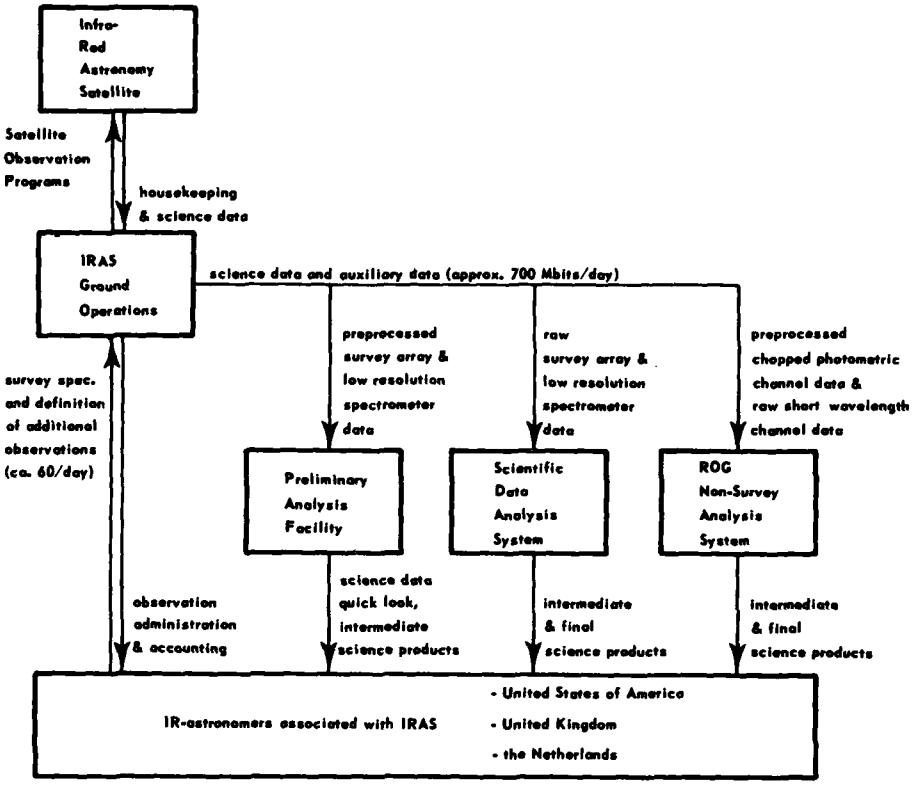


Fig. 7 Science information flow through the ground data system

time correlation data, is called the auxiliary data and is used by the science data processing centres to obtain, for instance, the required accuracy for telescope boresight pointing reconstruction.

The first recipient of the science and auxiliary data is the Preliminary Analysis Facility. It is the task of PAF to carry out a rapid assessment of the science data, and in particular the survey data, so that corrective actions can be initiated for the first following "12-hours" period. This fast response is necessary to meet the rigid requirements put on the repetition scheme for the survey scans by the astronomers, as this enables later processing to recognize and deal with unwanted celestial objects, like asteroids. The activities that are performed by PAF for the assessment consists of checks on the performance of the detectors of the survey array and low resolution spectrometer (noise output, detection statistics, total flux, radiation hits and response to calibration sources) and comparison of the detections with already known IR-sources. If these checks indicate anomalous behaviour of the instrument, an inspection of the data is possible using diagnostic tools provided by PAF. Next to this instrument oriented, mission critical, analysis, PAF also supplies an intermediate scientific data base with the necessary analysis tools, which will allow the IR-astronomers to assess the quality of the survey and guide the additional observations. The database will comprise the history of detections of a number of pre-selected sources and areas, as well as of point-sources in the longer wavelength bands and of "compact" sources (i.e. sources with dimensions in the order of the length of a detector, 4 arcmins). Finally, a database will be compiled of the background data that remains after subtracting point- and compact sources from the raw data. As only the low frequency components have to be retained, the amount of data is reduced by a factor of upto 8, using a two-dimensional spline-fit method, developed at NLR.

The raw science data and the auxiliary data are sent to SDAS over a 56 kbps wideband datalink, provided by NASCOM, requiring twice daily transmission of about 3 hours. Like PAF, SDAS also only processes the data obtained with the survey array and the low resolution spectrometer, but now with the objective to provide the science community with data products, that have been arrived at, after applying the projects best knowledge of spacecraft and instrument behaviour. The main task of SDAS is to produce from the survey data a catalog of infra-red sources, that meets certain strict requirements concerning reliability, completeness, sky coverage, and accuracy in photometry and position. For additional observations, that have obtained data in a mode similar to the survey scans, SDAS will use modules for the investigator concerned. For the deep-sky-survey observations SDAS produces a co-added matrix of grid cells for each of the four wavelength bands, for further analysis by the investigator. The major activities that SDAS has to perform to arrive at these data products are:

- telescope boresight pointing reconstruction
- determination of the transfer function of the instrument, using the calibration data gathered throughout the mission
- applying calibration data to the raw science data
- detection of the various types of sources (point-, compact- and extended-sources)
- correlation and confirmation of the detections on the various timescales (seconds, hours and weeks), to get rid of asteroïds etc.

The development of the software for processing the low resolution spectrometer is carried out by ROG at JPL. SDAS receives approximately 700 Mbits every day. Although no near real-time, mission critical, requirements are placed on SDAS, it is clear that with these amounts of data the development of backlogs

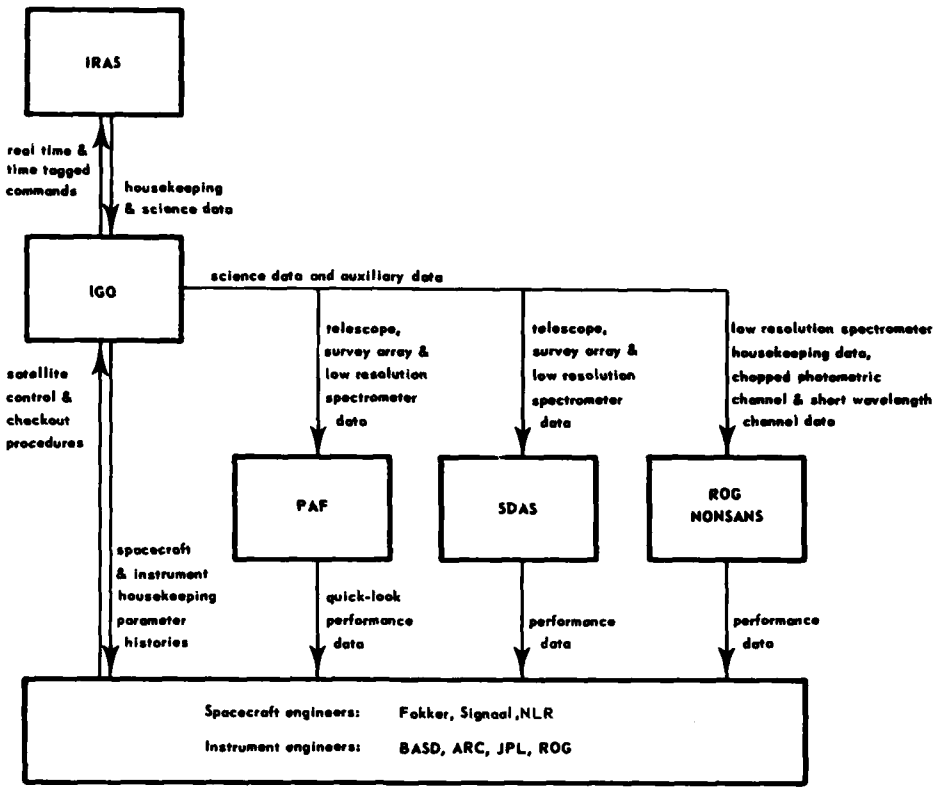


Fig. 8 Engineering information flow through the ground data system

can be detrimental. One of the major design goals therefore is to process 24 hours worth of satellite data within 16 hours of processing time. All products have to be delivered within 6 months after the end of the operations.

The ROG Non-Survey Analysis System processes all data obtained with the chopped photometers and the short wavelength channel. During operations, the processing for the chopped photometric channels will mainly consists of synchronous detection of the instrument data and merging with information obtained from the auxiliary data provided by IGO (like satellite position and telescope pointing direction). The data from the short wavelength channel will also be merged with information from the auxiliary data and archived.

6.2 Engineering information flow

The interface between the four major elements of the ground data system are the same for both the science and the engineering data flow, being the science and auxiliary data streams (ref. figure 8). The output, of course, is different. As an intermediate step of the science data processing a number of instrument characteristics are determined which will provide the instrument engineers with an indication of the performance. No specific processing is currently foreseen, to determine the instrument performance otherwise, neither at the science data processing centres, nor at the institutes of the engineers.

If the instrument performance data indicate anomalous behaviour, the engineers can perform a further analysis of the housekeeping data. IGO maintains a full history of all housekeeping data stored, with sampled tools for making plots, performing statistical processes, out-of-limit checking etc. During operations a number of telescope experts will be resident at the Operations Control Centre. The ground data system design, however, does not preclude the use of this data and these tools by engineers in the United States. To this end, both JPL and ARC will have a terminal facility linked with the OCC computer, via NASCOM lines, providing them with the same possibilities an engineer would have at the OCC. Another possible source for further analysis would be the raw science data itself, available at the OCC and at JPL. Access would be provided through the diagnostic tools of PAF and SDAS.

The responsibility for determining and maintaining the performance of the spacecraft during routine operations has been delegated to IGO. To transfer the necessary knowledge, the spacecraft engineers have to provide an extensive set of documentation comprising of unit descriptions, command and telemetry descriptions, and finally, control and checkout procedures. This documentation has to be kept up-to-date through the various phases of integration and test of the hardware. In addition IGO will send members of the operations team to observe and, where possible, to participate in the important test phases. During the launch and in-orbit-checkout phases, the project will station a number of spacecraft engineers at the OCC, to support the performance verification. They will also be available on call for IGO if a spacecraft emergency develops during routine operations.

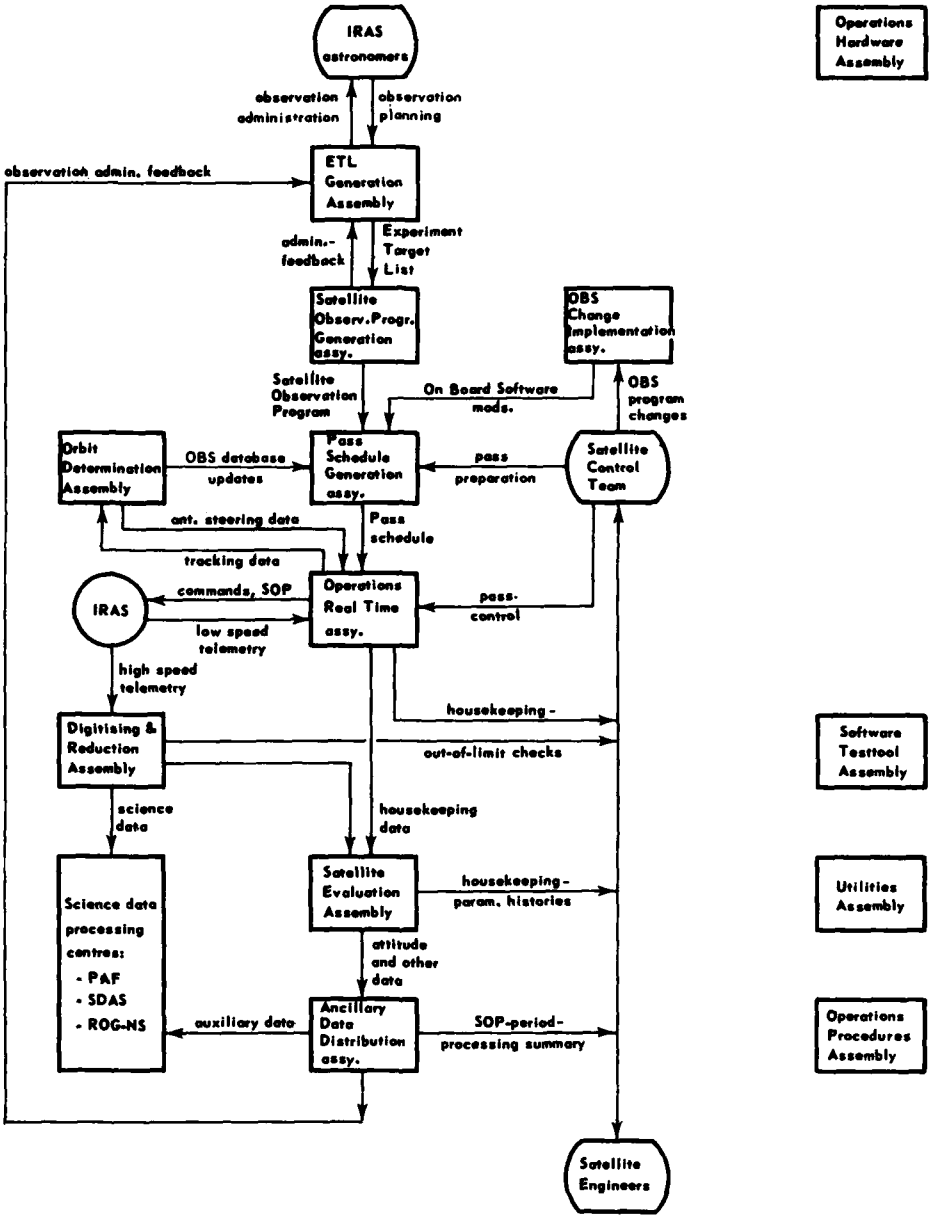


Fig. 9 IRAS ground operations

7. IRAS GROUND OPERATIONS

A more detailed description of the IGO part of the ground data system now follows. The following tasks have been allocated to this subsystem, viz:

- orbit determination and prediction
- provision of tools for the IR-astronomers to generate Satellite Observation Programs
- real time operations
- satellite health assessment
- spacecraft performance evaluation and maintenance
- data pre-processing and distribution to the science data processing centres
- filing of the raw data, from which the distributed data are derived.

The design of IGO identifies thirteen assemblies: eleven for the software section, one for the operations hardware and one for the development of the operational procedures. Figure 9 gives a schematic diagram of these assemblies and the flow of information through the IGO subsystem. It should be noted that only the important interrelations have been depicted, since a complete interface diagram would be incomprehensable and not serve any purpose. Also obvious interfaces have been deleted, like the link between every software assembly and the hardware assembly.

7.1 ETL Generation Assembly, EGA

The diagram starts at the top with the ultimate users of the system, i.e. the astronomers associated with the project. They can enter their requirements for the survey and the additional observations programs in a system that generates Experiment Target Lists (ETLs) for every "12-hours" period. Each ETL is a time-sequenced list of observations and is input for another program that converts the ETL into an actual Satellite Observation Program. Ideally, there is a one-to-one correspondence between an ETL and the derived SOP. The ETL basically is a human interpretable version of the SOP, the latter being nothing but a string of coded 16-bit words for interpretation by the on board software. In practice there will be differences between an ETL and the SOP, either because the SOP Generation software inserts operational manoeuvres (e.g. to avoid crossing the moon in between two successive observations) or because it expels observations which would violate satellite constraints. The ETL Generation software does check most of these constraints as well, but it must refrain from, or approximate, some CPU-intensive checks. This is the direct result of the basic philosophy that all software programs, that interface with the multitude of IR-astronomers, must be of a self-explanatory, interactive, nature. It was feared that, otherwise, the tools offered by the system would not be compatible with the majority of users. Interactive systems can only be operative if the response time is reasonably fast and, hence, CPU and I/O usage are kept low. The result of the SOP Generation process is fed back to the observation administration and accounting system also provided in the ETL Generation Assembly. Together with the feedback from the post-pass processing software, it presents the investigators a full history of all scheduled and future observations. It also offers the possibility for recording the scientist's assessment of the success of the observations executed.

7.2 SOP Generation assembly, SOG

The SOP Generation assembly must ensure that the observation programs ultimately produced do not endanger the satellite in any way. It performs checks on:

- satellite attitude, both during the observations and during the manoeuvres in between, to:
  - preserve the helium (i.e. not pointing towards the sun or earth)
  - keep the telescope clean (not pointing into the direction of the orbital velocity vector for extended periods, to minimize the collection of dust particles)
  - prevent blinding the instruments, by keeping the telescope viewing axis sufficiently away from the moon and other bright celestial objects
- satellite configurations, e.g. to verify that during survey observations the chopper of the CPC is switched off, as it causes significant interference with the survey detectors
- commanding through the SOP, to test whether the commands are allowed for the current satellite status and whether no timing constraints exist with respect to the previous command(s)
- possible power constraints, for instance, during eclipse periods
- position of the satellite, e.g. with respect to zones of high radiation like the South Atlantic Anomaly.

The latter example does not really endanger the satellite, but it could affect the scientific value of the observations concerned. Such observations are only flagged, but not expelled from the SOP.

7.3 OBS Change Implementation assembly, OCI

Experience with the previous Dutch astronomy satellite, the ANS (Astronomical Netherlands Satellite), which was also equipped with a re-programmable on board computer, has taught that post-launch modifications of the on board software are inevitable. The changes for ANS ranged from correcting certain attitude control modes (e.g. to allow for the non-nominal orbit it was launched into) to the creation of new data collection modes requested by the scientists. To allow this for IRAS, one of the original program development stations for the on board software will be transported to the OCC, together with all the relevant test-software packages (the OBS Change Implementation assembly). This, combined with the fact that one of the on board software designers has been added to the IGO team, will provide sufficient opportunity to maximize the return of the mission.

7.4 Pass Schedule Generation assembly, PSG

The Pass Schedule Generation assembly collects the pertinent SOP and any required modification to the on board software from OCI or, in case of standard database updates, from the Orbit Determination Assembly (like earth magnetic field model parameters, or the expected starttimes of eclipses). PSG compiles all this in on board memory load messages, conform the requirements for the uplink. It then draws up a schedule for the activities to be performed during the pass, such as the playback of the taperecorder, the dumping of the solid state memory and the loading of the SOP. Finally, ground station information is added for the benefit of the real time software: whether the prime station or a STDN station is going to be used, the expected times of arrival and departure of the satellite, etc. Most of the information for a prime pass can be generated automatically. The satellite control team has the possibility to manually create or adapt pass schedules for non-standard operations/passes.

7.5 Operations Real Time assembly, ORT

Prior to every pass a data flow test is run to test if all hard- and software elements are working properly. For every conceivable hardware item in the real time loop there is a redundant unit, including the processor, but with the exception of some parts of the prime antenna. So, in the majority of cases, at the end of the data flow test, an operational system will be available to take the pass. The pass duration is only in the order of 10 minutes. An average SOP load requires about 2 minutes, a solid state memory dump 4.5 minutes and a recorder playback 6 minutes. All these operations can be performed in parallel, with the exception of the dump of the SOP area in the on board memory. The SOP area has to be dumped prior to SOP loading to help explain possible mishappenings in the data gathered in the previous "12-hours" period. When SOP loading is completed the area is dumped again to check whether the SOP arrived correctly. After any required correction the on board computer can then be instructed to start the execution of the SOP.

Part of the solid state memory data is the program memory itself. All instructions are checked in real time by the Operations Real Time software and if discrepancies are detected between the ground-based version of the on board software and the program memory contents, the satellite control team is alerted. This would normally result in a delaying of the enabling of the SOP execution, until the cause of the discrepancy is discovered. If necessary, use can be made of the pass in a next orbit over the prime station, and if the situation seems sufficiently serious further emergency support from STDN can be requested. A complete reload of the on board software would normally require two consecutive passes, as the time required for loading and subsequent dumping of the program memory contents is in the order of 12 minutes.

7.6 Digitizing and Reduction Assembly, DRA

The housekeeping data stored in the solid state memory is not limit-checked in real time by the ORT software, but immediately after the pass. The high speed telemetry stream is also not processed in real time. During the transmission it is recorded by two redundant wide-band instrumentation taperecorders. The data is digitized after the pass by the Digitizing and Reduction Assembly and during this process the housekeeping parameters present in this stream are limit-checked. Both these limit checks will have been finished within 30 minutes after the end of the pass. If the checks indicate anomalous behaviour of trends on board the satellite, it is possible to setup the operations system for the pass on the next orbit. The satellite is visible on 2 to 3 consecutive orbits. The first one is normally taken as the prime pass. The others would in general not be taken unless circumstances necessitate it.

When the high speed data is digitized by DRA, possible data overlaps (e.g. due to repeated playbacks) are removed. In a next process, the data are decommutated, decompressed from 8 bit- to 16 bit-words and then a source detection algorithm is applied. After removing the detections from the original detector data, a B-spline convolution is applied on the remaining background signal to generate data for surface fitting. The output of the process, being the decommutated decompressed detector data, the detection-data and the B-spline-coefficients, is then forwarded to the Preliminary Analysis Facility. For the ROG-NONSANS, the detector data of the chopped photometric channel is decommutated and synchronously demodulated. The reason for doing this pre-processing in IGO for the science data processing facilities is that IGO has an extremely fast processing device connected to the computer system used for ORT and DRA. This device, an AP 1203 array processor gives, for this type of application, a considerable gain in processing time compared with other computers available for this project. It must be pointed out that the pre-processing of the survey data is performed with the goal in mind that PAF must be able to assess this data before the next 12-hours period is due. To make this possible, no account can be taken of e.g. the latest instrument calibration data dispersed through the observations. For this reason SDAS does not use this pre-processed data, but the original raw detector data.

7.7 Satellite Evaluation Assembly, SEA

Both ORT and DRA extract the housekeeping data, contained in the two telemetry streams and pass them onto the Satellite Evaluation Assembly. Upon receipt of the data all samples of the parameters are again limit-checked. ORT and DRA can only guarantee to make a check on every 2 to 3 samples, but not on each individual one, as they both are real time programs and the out-of-limit check does not have the highest priority. The housekeeping data stored in the solid state memory is actually a snapshot of all housekeeping parameters in the low speed telemetry at the time of storage, with all multiplexed channels decommutated. The amount of memory available for this storage (8 kwords, 16 bits) allows 24 hours worth of data to be stored with an interval of 20 minutes. The interval rate and storage capacity are easily adjustable parameters of the on board software. The data gathered during every pass are added to a history file, allowing engineers to evaluate trends from the start of the operations onwards. It is intended to keep this file readily available on disc, providing an almost immediate access. The housekeeping data stored by the recorder is much more massive, in the order of 35 Mbits per day. In fact, for some parameters it gives a sample frequency of 1 Hz. For every "12-hours" period all this data will be converted to engineering units. The storage space allocated on the OCC batch computer will allow 24 hours worth of data to be kept on disc in this format and hence, make it almost immediately accessible. Samples of this datastream, with a sample rate of the same magnitude as used for the parameters in the solid state memory, will be maintained on disc for a period of at least one week as well. All the data from this stream will finally be filed and remain accessible until the end of the mission, but not as easily as for the solid state memory data. Discussions with the spacecraft and instrument engineers have not indicated that this approach would be inadequate. Of course it will be possible to add a more comprehensive history file on disc for a limited number of parameters, if the need arises. The processing and display options of this assembly are generous and comprise, among other things, limit-checked parameter histories on display or printout, delta-check prints, and plots of parameter(s) versus time or versus other parameters.

Besides the housekeeping data samples, the on board software can generate two other types of data for storage in the on board memory. One of these is the event-related data, which consists of self-identifiable varying datasets that are stored at specific occasions, such as the start or the end of an eclipse. It is further used to store the data related to star-crossings over the detector slits of the starsensor accommodated in the focal plane of the telescope. These attitude calibration data are necessary for pointing reconstruction on the ground, but also to determine slowly varying parameters used in the on board attitude control algorithms (e.g. drift and scale factor of the gyros). These parameters have to be updated every few days to ensure a correct attitude of the satellite. A number of other attitude control related parameters, varying more slowly (in the order of months), require special purpose storage of certain parameters, sometimes coupled to specific attitude manoeuvres, so that their values can be verified on the ground. For this type of application the third type storage was devised: the Special Data Storage. This one is controlled through the SOP and can easily be varied according to the needs of the moment, both as to the choice of parameters to be stored as well as the storage frequency. Both the Event Related Data and the Special Data are dealt with in the Satellite Evaluation Assembly by a number of special purpose software packages. This variety in data storage methods is expected to be sufficient to perform all non-routine checks during the in-orbit-checkout phase of the operations as well.

7.8 Ancillary Data Distribution assembly, ADD

The attitude calibration event data are passed onto the Ancillary Data Distribution assembly. There they are tested and correlated, and made available as part of the auxiliary data for the science data processing centres to perform their pointing reconstruction. The auxilliary data further contain:
- orbital parameters
- satellite time versus TUC correlation
- satellite temperature measurements, that are only available through the low speed telemetry stream
- SOP administrative data
- observation administrative data
- attitude control algorithm parameters
- misalignment data of the attitude sensors.

A second task of this assembly is to compile a summary of the processes applied to the two datastreams and indicate the occurrence of anomalies, like the detection of out-of-limits or irregular event data. Finally, it provides the observation administration in the ETL Generation Assembly with a feedback on the technical success of every observation scheduled for the past period of 12 hours.

7.9 Orbit Determination Assembly, ADA

Like the previous assembly, the Orbit Determination Assembly interfaces with nearly every other assembly. As the name indicates, its prime function is to determine the current orbital parameters, and from these predict the orbit for the rest of the mission. The altitude of the IRAS orbit, 900 km, does not necessitate the use of systems like Doppler or range/range-rate for this process. In fact, the satellite itself does not carry any special purpose hardware for orbit determination. The program uses the antenna angle measurements of the tracking stations, which are logged by the Operations Real Time software. From the predicted orbit a number of other items are derived, such as the time of the station passes, the satellite track over the station, the times of entering and leaving the zones of high radiation, the start and end times of eclipses, etc. It further compiles the database parameters for the on board software mentioned earlier (parameters of the earth magnetic field model, eclipse warning times and others) and computes the ephemerides of the celestial bodies, that have to be taken into account by the ETL and SOP generation assemblies.

7.10 Software Testtool Assembly, STA

For an integrated test approach of the IRAS Ground Operations subsystem a number of simulators have been defined as part of the Software Testtool Assembly. A Real Time Simulator is being developed for checking out the ORT-software. It generates low speed telemetry data, taking into account the commands uplinked. Some "fiddling" will be possible, to create anomalies, such as out-of-limits, failure to respond to commands, parity errors. As this simulator is designed to test out the real time operations, it will not provide realistic contents for the data which are only processed post pass. This data, the high speed telemetry and the information stored in the solid state memory, are dependent on the SOP loaded. They will be generated by the SOP-dependant Data Simulator, based on actual SOPs and simulated science data. The latter is obtained from the Infra-Red Telescope Simulator, devised by ARC, that can simulate the data of the survey array for a certain sky-input. Also the SOP-dependant Data Simulator will allow a certain amount of fiddling, for instance to create data-overlaps and -outages. Another testtool is the SOP Generation Simulator, which will enable the creation of SOPs based on a manual input rather than from the observation planning facility. It allows a reasonable uncoupling of the development schedules of the Pass Schedule Generation/Operations Real Time software and the observation planning facility. It further permits to make a SOP that correlates with the data obtained from the ARC simulator, which can be of benefit for testing the post-pass processing systems.

7.11 Utility Assembly, UTA

The last software assembly is the Utility Assembly. It is a collection of subroutines, programs and macros which are not specific to a given program or assembly, or which are of a general nature. It gives routines for word- and bit-manipulations, communicating with the operating system, handling labelled magnetic tapes, copying disc-files to tape and vice versa, obtaining the current data and time and many others. These utilities are also available to the Preliminary Analysis Facility.

7.12 Operational Hardware Assembly, OHA

The eleven software assemblies described above are implemented on the following computer systems:
- a dual PDP 11/34 configuration, each processor having 80 kwords of MOS memory and 14 Mbytes of disc storage, and sharing a 176 Mbyte disc and an AP120B array processor; the software of ORT and DRA reside in this system, together with a small part of the Pass Schedule Generation assembly
- a Philips P856M system, originally used for the development of the on board software and now accommodating a part of the OBS Change Implementation assembly
- an ICL 2960 system, with 1 Mbyte of memory, 4 200 Mbyte discs, 6 tape units and a host of VDUs, plotters and (interactive) graphic devices, accomodates all the remaining software packages of IGO.

The ICL 2960 system is a project dedicated computer, and is shared with the Preliminary Analysis Facility and the ROG Non-Survey Analysis System. It has two dial-up lines, which will allow scientists and engineers from either side of the Atlantic to use the facilities at the OCC. It will be possible to use standard NASCOM lines, instead of the normal pay-phone, to enter the system from JPL or ARC. File transfer over the wideband data link from the OCC to JPL/ARC will be routed through the PDP 11/34 system, as one of the off-line tasks of the Operations Real Time assembly. In view of the rather large amount of data to be transferred between the real-time and the batch computersystem (see also fig. 10) an inter-computer link will be installed.

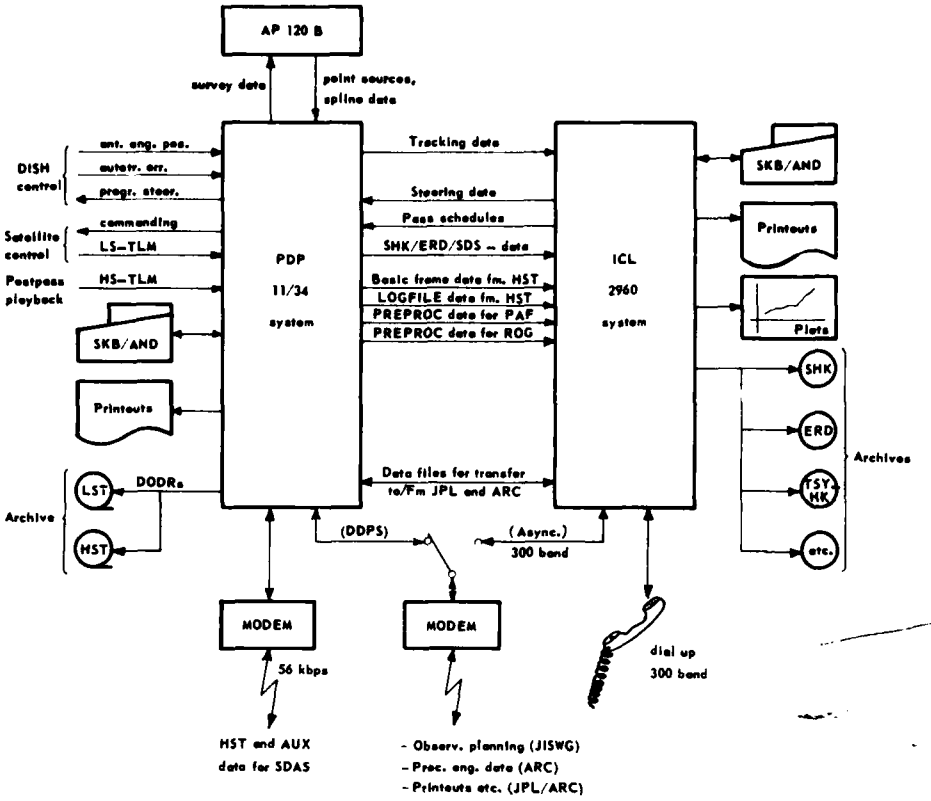


Fig. 10 Use of the OCC real-time and batch computer system

The Operations Hardware Assembly (OHA) consists of the aforementioned computersystems, the remaining control centre hardware (bit- and frame-synchronizers, wideband instrumentation taperecorders, NASCOM line interface equipment, telexes, etc.) and the tracking station (parabolic 12m dish, antenna drive system, receiver and exciter system, etc.)

7.13 Operations Procedures Assembly, OPA

The other non-software assembly is the Operations Procedures Assembly. Its output is a set of operational documentation for the satellite control team:

- Flight Operations Procedures Manual, gives the detailed procedures covering software and hardware control (both for flight and ground systems) and the reporting requirements to be followed during satellite operations
- Satellite Control Handbook, provides a description of the satellite subsystems and units to serve as a reference for the Flight Operations Procedures Manual
- Format Control Handbook, describes all human interfaces of the data system at the OCC, i.e. input formats, lineprinter output, display formats, etc.

This set, together with the project provided satellite documentation and the documentation provided by the software- and hardware implementers of IGO, will allow the satellite control team to execute its tasks in a largely independent way. It is clear that, whenever unforeseen anomalies arise, they can request assistance from the satellite- and/or ground data system experts. In view of the widely spaced locations of the participants in this project, this ought to be kept to a minimum. For the documentation it means that it has got to be rather detailed and definitely up-to-date, and to incorporate the expertise gained during test and integration of the systems. A secondary task of this assembly is to generate documents to familiarize new operators with the project and to assist scientists and engineers visiting the Operations Control Centre in finding their way.

8. OBSERVATION PLANNING FACILITY

As a typical example of a complex software package within the IGO subsystem the observation planning facility will now be described in somewhat more detail. The amount of observations that have to be defined to arrive at the goals of this mission is quite staggering: roughly 10,000 for the survey program and 20 - 30,000 for the additional observations. The generation of the survey observations is further complicated by the strict repetition scheme for the coverage, imposed by the astronomers. It was felt that the ground data system, in casu IGO, should provide an observation planning facility, that would relief the astronomers from a heavy routine involvement but would still allow them the greatest amount of control.

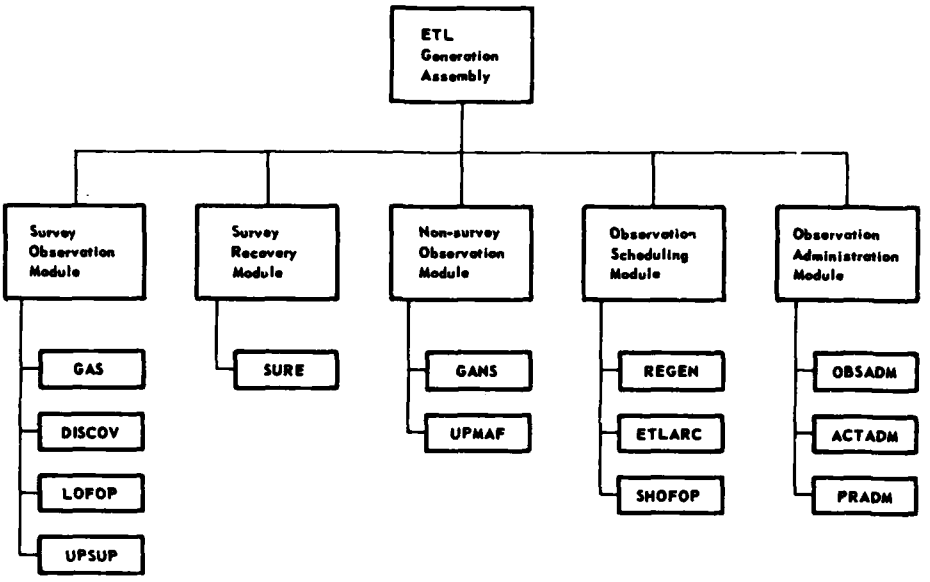


Fig. 11 ETL generation assembly breakdown

The basic requirements on the observation planning facility for IRAS can be summarized as follows:
. to generate survey scans automatically, according to a strategy of which the parameters can be adjusted by the astronomers, but remaining within the satellite constraints
. to provide tools for defining special scans ("recovery scans") for survey observations, which failed to satisfy the requirements of the astronomers
. to facilitate the definition of the additional observations by the investigators
. to provide tools for efficient scheduling of the additional observations in between the survey observations, minimizing the time lost in slewing from one observation to the next.

Each of these requirements is reflected in one of the five subassemblies of the ETL Generation Assembly (fig. 11):
- the survey observation module
- the survey recovery module
- the non-survey observation module
- the observation scheduling module
- the observation administration module.

The last module in this list covers the derived requirements on observation administration. In the case of IRAS, the administration system not only has to keep track of the information about scheduling and success of the observations, but also has to maintain an "account"-system. The amount of time available for additional observations must be distributed fairly over the science communities in the different countries. At the same time, duplication of effort by the different communities must be avoided in order not to waste the precious satellite time.

8.1 Survey Observation Module, SOM

The Survey Observation Module comprises four individual programs:
. GAS, the Generation Aid for the Survey, produces the actual observations in a format which is acceptable for the scheduling software
. DISCOV, displays the planned survey coverage by means of an Optronix Photowrite device
. LOFOP, produces the "long-term forecast", which is a lineprinter listing giving the details of the observations generated by GAS for the orbits requested
. UPSUP, for updating the parameters of the survey strategy, as requested by the astronomers.

In the following, we will only deal with GAS. The others are support programs to allow the control and monitoring of the processing in GAS and do not contribute to the basic operation of the observation planning facility.

The functions of GAS are the following: to
. (for a specified period in the order of months) determine survey scans according to a parameterized strategy, taking into account previously planned sky coverage
. take into account restrictions with respect to satellite attitude and position and the occurence of events, such as eclipses and station passes
. allow reservation of orbits for special operations and survey recovery
"format" the scan into observations, for further scheduling
. allow resetting and rerunning for a specified period with new parameters for the strategy.

The survey strategy parameters determine the coverage redundancy, the repetition rate, the coverage rate and a number of "soft" constraints, like the minimum scan length and the radius of the area to be avoided around the moon. It is possible to maximize the coverage rate, resulting in an initial sprint as far as the attitude limits will allow. For the long term mission planning it means a rapid coverage of about half the sky, which then has to be followed by a much slower coverage, governed by the precession of the orbi-

tal plane. The main disadvantages of such an approach are that operational hick-ups are rather detrimental to the planning and that additional observations become more clustered in time and, hence, in sky position. Another set of parameters will provide a much more even coverage rate throughout the survey. This strategy frees a regular amount of time for additional observations or, in case of operational anomalies, for survey recovery. It is clear that such a strategy is to be preferred from a routine-operations point of view, although it does provide less coverage initially. An early satellite disaster could then in principle result in a smaller return from the mission. In practice, the difference in coverage is not expected to be that large as, operational problems (like station-outages) are likely to occur especially in the beginning of the operations.

8.2 Survey Recovery Module, SRM

It must be realized that GAS is designed to optimize the long-term survey planning. When the survey observations have been executed, an indication of their success is fed back to the Survey Administration File, created by GAS. If there are large discrepancies between planned and actual coverage it is possible to reset GAS and produce a new set of survey observations for the upcoming month(s). This would not be very efficient for small anomalies, such as the loss of data of part of a scan due to a short telemetry outage during the playback of the recorder. To allow a rapid correction of such small discrepancies, with the least disturbance to the long term planning, the program SURE of the Survey Recovery Module has been included in the design. It makes use of the gaps between the survey observations and will generate recovery scans for the failed portions of the original survey scans. Contrary to GAS, SURE is an interactive program to allow the greatest flexibility in specifying when and how to recover failing survey portions.

8.3 Non-survey Observation Module, NOM

The Non-survey Observation Module consists of two programs:

- GANS, the Generation Aid for Non-survey observations, which produces the additional observations in a format acceptable for the scheduling software
- UPMAF, which allows updating of the macro-file used by the scientists when defining additional observations through GANS.

GANS is the tool with which any authorized astronomer defines his additional observations for inclusion in the Experiment Target Lists and, ultimately, the Satellite Observation Programs. It is an interactive program that enables the astronomer to specify the sequence of attitude and data storage modes for a specific observation, and the necessary commands to configure the instruments. He does not have to specify the visibility window of the observation, as the program will calculate that for him. Only if, for specific reasons, he wants to restrict the scheduled window he can specify a range of orbit numbers (or a time period) and the limitations within the orbit. In so far as possible at this stage, the program will check the requested sequences on their feasibility with respect to the constraints mentioned earlier and, if necessary, will inform the investigator of any problems. To speed up the interactive process for series of observations that have certain characteristics in common, a macro-facility is included for the experienced user of the system. It allows the astronomer to specify and use his own set of macros, or use the IGO-provided standard ones. The NOM has also a built in feature for the satellite control team, with which they can specify operational manoeuvres, that will be required for satellite checkout.

8.4 Observation Scheduling Module, OSM

The observation generated by GAS, SURE and GANS need to be scheduled, within the satellite constraints, in such an order that they obey the directions provided by the astronomers, with respect to repetition rate, position in orbit, etc. On the other hand, as little time as possible should be waisted on non-productive manoeuvres as slewing from one observing position to the next. The prime function of the Observation Scheduling Module is to provide a tool with which the astronomers can schedule the survey and additional observations. The module consists of three software packages:

- REGEN, does the actual scheduling and produces the ETL
- ETLARC, files the ETLs that have been translated successfully into a SOP, that was loaded
- SHOFOP, produces the "short-term forecast", which is similar to the longterm forecast produced in SOM, but now includes the additional observations too.

The REGEN program can be run in a number of modes. Normally the program would first schedule the survey and survey-recovery observations according to the instruction generated by GAS/SURE. The remaining gaps are filled with additional observations, based on such criteria as the slew-time required, the remaining idle time, the priority of the observation, etc. The priority is dynamically allocated and is dependent on the initial priority given by the astronomers and the size of the remaining visibility window. In other words, observations approaching the end of their visibility window will receive a higher priority in scheduling. The program tries to find an "optimum" solution with respect to the idle time (slew time) of a complete ETL, within the constraints of the satellite. ETLs thus generated can be altered using the manual mode of the program, in which all the constraint-checking will still be performed.

8.5 Observation Administration Module, OAM

The Observation Administration Module keeps track of every observation defined. It notes whether an observation has been scheduled in an ETL/SOP, what its success was from a technical point of view and allows the astronomers to enter remarks on the scientific success. As stated earlier, it also maintains an account of the additional observation time, so that a fair distribution over the science communities of the three participating countries can be warranted. The module consists of three programs:

- OBSADM, maintains the actual observation histories
- ACTADM, derives from these histories the accounting information
- PRADM, provides a printed record of the histories or the account, upon request.

The information stored by OBSADM and ACTADM can be retrieved on various keys in an interactive manner, to increase the operational usefulness.

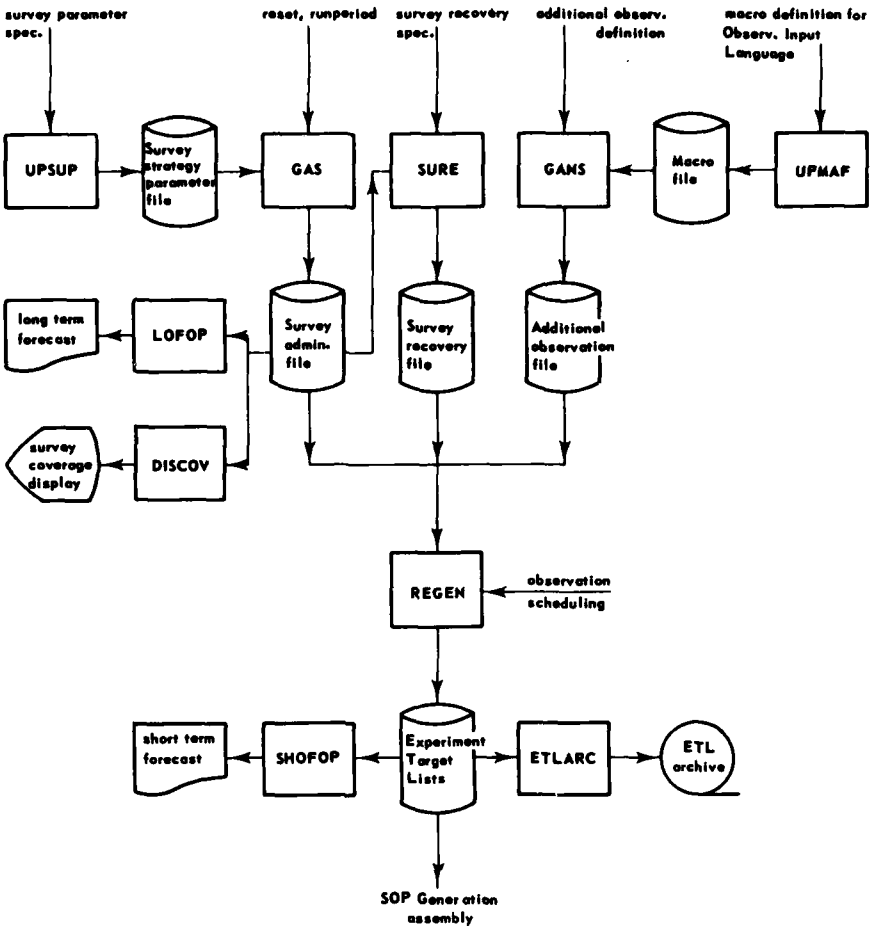


Fig. 12 Observation planning facility

Figure 12 shows the interrelations of the various software packages of the observation planning facility, excluding the administrative programs of the OAM.

9. DATA SYSTEM CONFIGURATION CONTROL

The size and complexity of the data system for IRAS, further complicated by the multi-national project setup, requires a firm configuration control mechanism. The documentation system necessary for this will be described, in particular again in somewhat more detail for the IGO subsystem (Fig. 13).

The overall mission objectives and requirements are given in the Joint Project Requirement Document (JPRD). The requirements placed on the data system are further elaborated into requirements for the on board and ground data systems, and finally allocated to the different subsystems, such as IGO. The JPRD also identifies the various Interface Control Documents (ICDs) that are necessary to control the interfaces between the subsystems. Each ICD describes the complete interface between two subsystems down to bit- and signal-levels. These ICDs are already drawn up in this very detail during the design phase. If the interface specifications cannot be met, as discovered during the implementation or test-phase, then a new interface has to be negotiated between the subsystems. It is the responsibility of the individual subsystems to ensure that such changes are properly implemented in their design.

The requirements that are placed on a subsystem through the JPRD and various ICDs are further elaborated and allocated to "assemblies" (different names are used by other institutes), as described in the subsystem Requirements Specification. Down to this level all documentation is controlled by the project. A change request for any of these documents must be sent to all institutes, even when they are not explicitly affected. The response of these institutes is then taken into account for the decision to allow the change to take place, or whether another solution is more appropriate.

For most subsystems of the data system, the Requirements Specification is translated into a Design Specification. This identifies the different programs within each assembly and spells out their tasks. At IGO the Design Specification is also used for internal control (i.e. not by the project, but by the subsystem itself) of the interfaces between the assemblies. If during program development an assembly interface cannot be met, authorization is required by the subsystem manager (or the software co-ordinator) for any proposed modification to this set of interface definitions. The assembly designers must ensure that such modifications get reflected into the design of the affected programs.

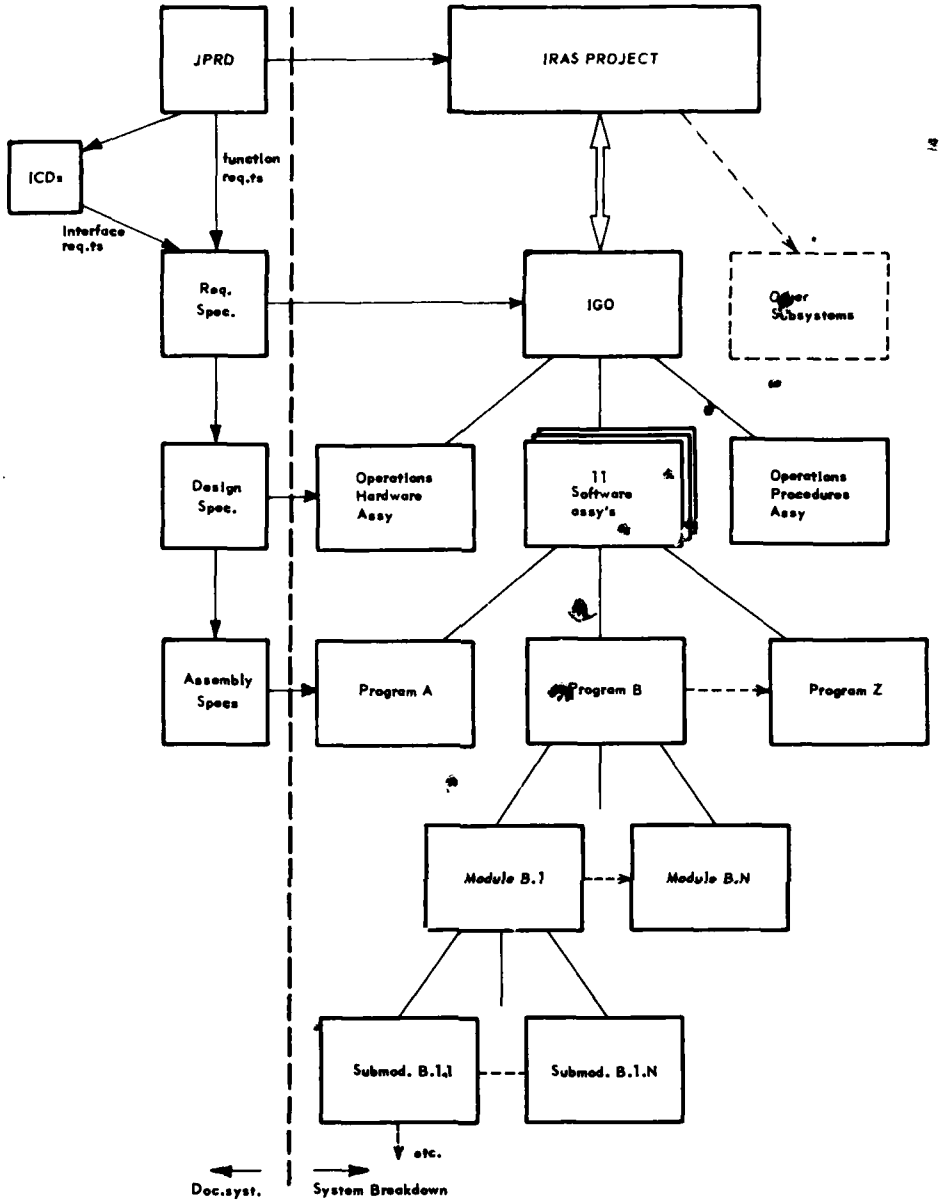


Fig. 13 Documentation system for configuration control

It is general practice within IGO that before actual coding of a program is started, first an initial paper-design is made. This is to ensure that the programs are well thought out, and thus operationally reliable, and do not develop into incomprehensible pieces of software in the course of the implementation. This paper-design is compiled into a set of Assembly Specifications and is kept up-to-date during the coding and test phase. Although such an overhead initially may seem unnecessary, one must not forget that the operational software needs to be carefully documented in the end anyway, to allow for post-launch maintenance. When it is already available during the implementation phase it allows verification of the design of critical programs in an early stage ("walk-throughs") and identifies complicated areas which may need a higher priority in the development. It is our experience that programs developed in such a manner exhibit an advantageous modularity and, from an operations point of view, are very robust and easy to maintain. Another small advantage is that user-manuals can easily be derived from this documentation

To ensure that the programs written meet their specifications every operational program has to pass an acceptance test. A test plan is drawn up for each program that correlates the test-activities with the requirements given in the Requirements Specification and ICDs, where necessary), and upon succesful completion of the test a test report is submitted. Once a program is tested and accepted, modifications are no longer allowed unless authorized by the subsystem manager or the software co-ordinator. Depending on the alteration, the program may have to be re-tested again for final inclusion into the operational system. A similar approach is followed for the integration of the various subsystems into a total datasystem. To this end an End-to-End Information System Testplan has been written, which has the concurrence of the individual subsystem managers and is approved by the project.

The ultimate proof of the pudding, before launch, is obtained by a number of simulations, that will involve all operational elements of the system, including the operators and the users. In view of the basic cycle of 12-hours, such a simulation must be of a rather long duration (in the order of a day or two) to create the expected operational environment. These simulations will start several months before launch with the prime elements of the data system, to allow possible tuning of the system with respect to the users. Later, they will include the remainder of the subsystem as well (such as STDN stations) to demonstrate the readiness of the total data system.

ADVANCED DESIGN CONCEPTS AND PRACTICES IN THE F-16 MISSION COMPUTER SOFTWARE

Judith A. Edwards
General Dynamics
Fort Worth, Texas 76101

Abstract - Improved system performance resulted from several broad-scope software design decisions that were applied to the mission computer for the F-16 avionics system. These design decisions were made in the areas of the control and scheduling functions. The selected control concepts relied upon a table-driven system that employed only positive logic and a simplified executive. The Multi-computer configuration was a loosely coupled in an asynchronous network. Communications in the network were implemented via a standard 16-bit protocol, MIL-STD-1553 [1]. Data consistency was adddressed in the basic interface specification as well as the detailed implementation. In general, the timing and hardware design features were hidden from the algorithms through system control approaches. In all, the flexibility and optimizations that resulted from the implementaticn concept have been well-established through two block updates and several demonstration and research programs.

Descriptors: asynchronous, table-driven, controllability, scheduler, dispatcher, DMA, re-entrancy.

The avionic system for the F-16 ([2] and [3]) is a distributed network of computers interconnected by a MIL-STD-1553 data bus (Figure 1-1). The mission fire control computer (FCC) serves as the integrating element, i.e., performs communication control and system-level computations. The software was developed by the prime contractor for two subsystems - the FCC and the stores management set (SMS). The functional partitioning and the subsystem interface data had considerable influence on the system performance and enhancement capacity throughout the program. The purpose of this paper is to outline the design concepts implemented in the F-16 mission computer and to outline the benefits realized.

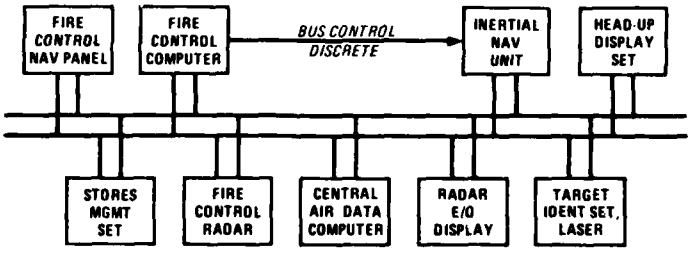


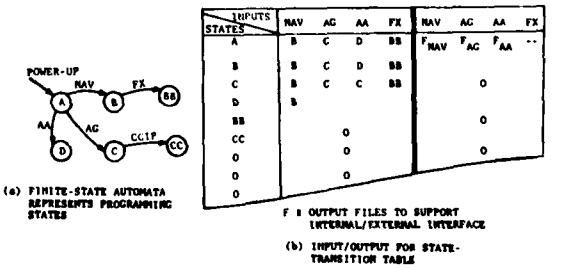
Figure 1-1 Avionic System Architecture

An important goal was to engineer software that would support integration efforts, change requests, and growth requirements. This was accomplished through the creation of a table-driven system - a system in which the algorithm evaluates the logic conditions to compute an index into a tabular data structure. The types of items in the data structure are control and data, which are needed at that particular logic state (see the example in Table 1-1 and the associated Figures 1-2a and 1-2b.) Such a structure localizes the impact of a change to the software data structures instead of to the algorithms. Also, the table-driven approach allows the designer to balance the task execution demands in the computer to remain nearly constant over a wide-range of pilot selected options. This approach also reduces the technical risks usually associated with real-time software in that the execution states of the system are deterministic. Additional benefits are derived by reducing the testing, documentation, revision, and subsystem, changes impacts.

		CASE: 1	2	3	4	5
CONDITIONS	NAV	Y	Y			
	AG	N				
	AA	N				
	FX	N	Y			
	CC1P	N				
	0					
	0					
	0					
ACTIONS	TASK A	Y	N			
	TASK B	N	N			
	TASK C	N	N			
	TASK BB		Y			
	OUTPUT F_{NAV}	Y	Y			
	OUTPUT F_{UP}	N	Y			

Y = YES
N = NO

Table 1-1 Decision Table



(a) FINITE-STATE AUTOMATA REPRESENTS PROGRAMMING STATES

(b) INPUT/OUTPUT FOR STATE-TRANSITION TABLE

Figure 1-2 Table-driven Systems Maps into Program States Based on Inputs

This paper is presented in six sections. Section 1 addresses the concept and design of the operating system. Section 2 covers the design of reliable network communications management. Section 3 provides a discussion of the software mechanization of algorithms and finally, Section 4 discusses the lessons learned during flight phases, research and developments studies, and special demonstration programs. A summary and bibliography are presented in Sections 5 and 6, respectively.

1. Design Considerations

The F-16 mission computer provides avionic system coordination for air-to-air, air-to-surface, and navigation functions. To meet these functional requirements, the FCC operational flight program is partitioned into components. Components are large-scale partitions, which generally consist of functionally related processes (e.g., see [4]), and are subdivided into segments, which are the smallest entity "dispatchable" by the operating system. Table 1-2 shows this partitioning of the F-16 operational flight program. The remaining subdivisions in the program are subroutines.

Component	Number of Segments	Size	Remarks
*Executive	4	279@	Interface with Interrupt structure Table-driven
*System Control	11	3251	Logic, Table-driven
*Bus Control	8	2250+	Peripheral IO control, channel programs, formatting, error checking Table-driven
*Initialization & Error Handling	6	531@	Error interrupt routines, Power up Initialization
Nav Support	5	910	Algorithms
Fixtaking	31	2720	Algorithms, Pointer-driven
Energy Management -Combat -Cruise	 4 5	 948 970	 Algorithms Algorithms
Air-to-Air Support	2	200	Algorithms
Air Missiles	3	1570	Algorithms
Air-to-Ground	14	2716	Algorithms
Stores Select	2	1702	Hash-coding, Pointer-driven
Data Entry of Display	1	2140	Table-driven, Pointer-driven
Self-Test	1	1546+	Table-driven
Support Utilities	N/A	706@	Numerical Analysis Subroutines (15)
Data Base of Stack Area		2433	Uses 2 Compools and 3 Stacks

Total Memory = 25262

Notes:

@ - All Assembly Language
+ - Some Assembly Language Segments
* - Operating System
x - algorithms use "based", or pointers, to access items in tabular data structures.

Table 1-2 Component Partitioning in Segments

1.1 Segment-level interface

Design aids were used to document the system segmentation interaction [5]. Figure 1-3 shows the control-flow and data-flow among segments of one component. A similar flow diagram among all components was prepared. It is important to note that the use of the logic and data in the design aids is not a "constraint" upon the component but is the interactions desired by the system engineer. These diagrams are then used to integrate the tasks with the operating system.

1.2 Algorithms

One of the earliest design considerations was to isolate the interface with the pilot from the application segments. Therefore, the applications segments were designed to be as "pure" as possible - that is, not to incorporate into the code mode-related or interface-related logic. This decision also avoided the embedding of detailed knowledge of performance characteristics of the subsystems into algorithms and improves flexibility and portability of algorithms and robustness to changes in external subsystems.

The algorithms were also not allowed to perform operating system functions, e.g., deschedule a task (including itself) or cause a task to enter a "wait-state" until a specific event occurs. Algorithms that were of excessive length were relegated to one of the background task queues, e.g., Kalman Filter, with dynamic extrapolation tasks in foreground. During task integration, all programs were placed in background queues unless absolutely required to be in foreground, i.e., time- or sequence-dependent. The criteria for allocation of tasks to foreground were (1) small resource utilization, (2) required for control of system, and/or (3) timely flow of valid data. All algorithms were required to be able to run in either type of task queue.

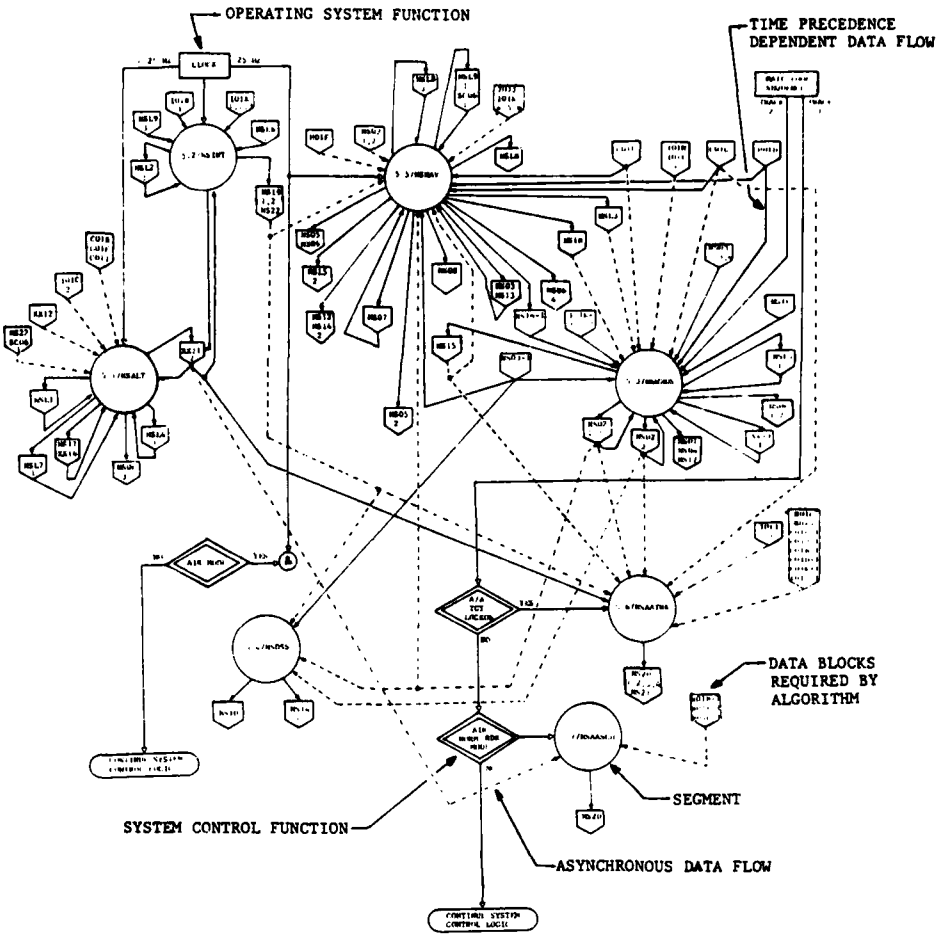


Figure 1-3 Structure and Data Flow of the Navigation Support Component

These early management decisions lead to a portable library of algorithms, which are generic for a large number of mission peculiarities. The modularity, reentrancy, and data utilization allow the software to be reallocated with few coding changes. Data sharing was well supported by JOVIAL data structures. The allocation and initialization of data were managed through the language and design-aids.

1.3 Operating System

The fundamental difference between real-time operating systems for avionics and for commercial systems is that, in the former, all tasks and communications are known "a priori". This knowledge greatly simplifies the design of the operating system. Large, general-purpose operating systems must by nature be complex, self-protective, priority-driven, and adaptive. They require many man years to design, develop, and maintain and typically must assume unknown task ensembles with potentially malicious subscribers, variable resource utilization, little coordination/cooperation among subscribers, and protection of I/O files from spurious faults. On the other hand, the "a priori" nature of the loading and data-transmissions greatly simplifies the functional requirements for the avionic operating system without loss of generality.

The F-16 operating system consists of an executive, a system control, a bus control, and an initialization and error-handling component. The executive portion includes the modules required to select and dispatch task-queues as a function of time and/or CPU-utilization. System control is the component that coordinates the algorithms, makes executive requests, and interfaces with the pilot. The communications are managed through a parallel bus control algorithm [6]. Initialization and error handling are machine-dependent modules and not of particular interest here.

1.3.1 Executive Component

The F-16 executive performs those functions required to manage dispatching periodic and background tasks. This component of approximately 280 words is one of the few written in assembler language. The executive determines the appropriate task-rate queue to be selected at each 20-msec minorframe. The system supports several background queues, but currently, only two are in use. These queues are selected on a percentage basis of CPU-utilization.

Each "timeslice", or periodic, task queue, which includes bus control execution requirements, are much less than the 20 msec minor-frame. Therefore, the remaining time is apportioned on the basis of the system mode processing requirements so as to assure timely completion of background algorithms.

1.3.2 System Control Component

One component, system control, was given the functional task to perform the integration and interface for the command/response mechanism. Fifteen modes were defined and implemented; an example of the mode structure is provided in Table 1-3. Basically, the modes were defined with positive logic (discussed in Paragraph 3.1) and implemented with a state-of-the-system concept taken from Finite State Automata Theory, [7]. The mode and option-dependent scheduling were described by means of a directed graph, Figure 1-4.

MODE NUMBER	MODE NAME	MODE ENTRY LOGIC	MODE DESCRIPTION
12	EO WEAPONS	wpn del enable & eo	o EO SYMBOLOGY o RADAR COMMANDED TO AIR-TO-GROUND RANGING o NO ENERGY MANAGEMENT DISPLAYS o ESSENTIAL NAVIGATION AND CONTROL FUNCTIONS
13	ALT CAL	(¬(wpn del en) | no weapon mode) & alt cal & INU good	o A/G TARGET DESIGNATOR POSITIONED ON STORED TARGET o RADAR COMMANDED TO AIR-TO-GROUND RANGING o CRUISE E/M AVAILABLE IF SELECTED o ESSENTIAL NAVIGATION AND CONTROL FUNCTIONS
14	FIX	(¬(wpn del en) | no weapon mode) & ¬alt cal & fix & INU good	o A/G TARGET DESIGNATOR POSITIONED ON STORED TARGET o RADAR COMMAND DEPENDENT ON TYPE OF FIX o CRUISE E/M AVAILABLE IF SELECTED o ESSENTIAL NAVIGATION AND CONTROL FUNCTIONS
15	BASELINE	(¬(wpn del en) | no weapon mode) & ¬alt cal & ¬fix	o A/G TARGET DESIGNATOR POSITIONED ON STORED TARGET o INTERRUPTIVE SELF TEST RUN ON SPECIFIED SYSTEMS o RADAR COMMANDED TO STAND-BY OR BUILT IN TEST o CRUISE E/M AVAILABLE IF SELECTED o ESSENTIAL NAVIGATION AND CONTROL FUNCTIONS

Boolean Expressions Definitions:

INU good = (attitude/reference fail | navigation fail | navigation data unavailable | digital attitude data invalid)

wpn del enable = store ready | simulate

vip enable = vip & ((ccip & rocket) | dive toss | beacon | ccrp | ladd)

no weapon mode = (dogfight | missile | snapshoot | lcos | ccip | dive toss | ccrp | strafe | beacon | ladd eo)

fix = radar | tacan | hud | visual overfly

& = and

| = or

¬ = not

Table 1-3 System Mode Definition (4 of 15 Modes)

The table-driven approach, described earlier, evolved from decision-tables and data structures techniques [8]. Other components applying these table methods are fixtaking, bus control, executive, data entry, stores select, and self-test. The implementation of these concepts was easily supported by the features in JOVIAL J3B language. System control component assures smooth and timely transitions. Design requirements were imposed upon the display processing to prevent flutter, blinking, or jittery symbology. Erroneous information was also prohibited by the specification requirements and assured through the state changes managed by system control.

1.4 Operational Characteristic

The operating system must function as a coordinated set of tasks. This coordination must guarantee that scheduling anomalies and race conditions do not arise. One design aid that shows control and data flow for cooperating processes is the Petri net [9]. Figure 1-5 is the Petri-net for the F-16 operating system. For an interrupt-driven system, this aid provides visibility into the system performance. The performance of the task queues is discussed in Section 4.

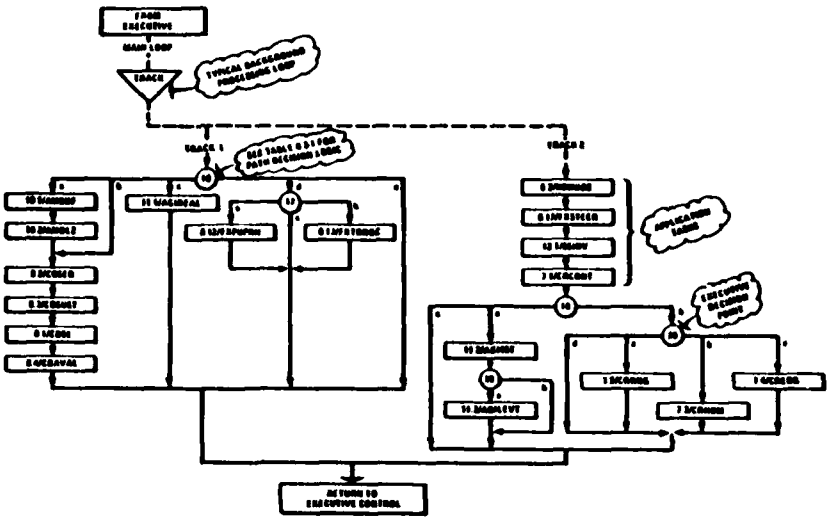


Figure 1-4 Digraph of Executive Sequence Serves As Design Aid

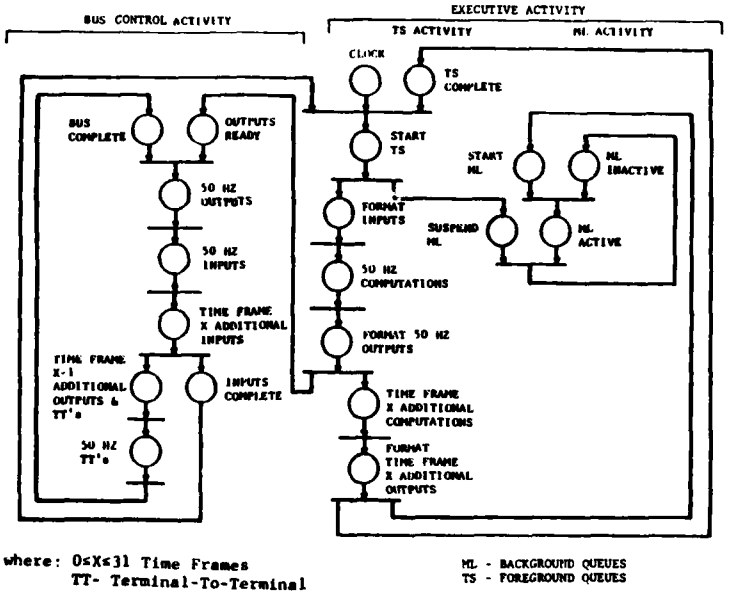


Figure 1-5 Representation of FCC OFP Operating System as a Petri Net

2. Reliable Communications

Several design considerations which contribute to reliable communications, arose during the definition of the interface. One of the first considerations was to recognize that the avionics multiplex data bus is not an infinite resource. The coordination of communications was facilitated by simple periodic transmissions and "time-tagging" data blocks. Data consistency was assured by the bus control algorithm. A "loosely coupled" design approach was implemented to improve the reliability of a multiple computer configuration. The primary benefits are that subsystem problems are not propagated through the system and that the need for coordination of subsystems is lessened. Presently, F-16 subsystems have no requirement for synchronization. However, the system was flexible and this feature could easily be accommodated by the table-driven bus control algorithm.

2.1 Periodic Transmissions

As a result of the interface definition, the majority of the transmissions for the F-16 avionics are periodic. The designers recognized that the bus control algorithm could perform periodic transmissions in a reliable straightforward manner. This concept lead to a concurrent management of the DMA processes. Communications do not become degraded in order to support a large number of nonperiodic transactions. Special events may lead to alterations in the command chain, which are easily accommodated by the updating of the command links. This approach gained the mission computer a balanced number of transactions within any minor frame. Such balance is necessary in order not to degrade the computational rates of background task queues.

2.2 Data Consistency

In a sampled data system such as a real-time avionics system, data consistency is a primary concern. In an aircraft with the F-16's dynamics capability, data inconsistencies can lead to wide fluctuations in sighting symbology, erroneous display information, and poor subsystem coordination. This situation is easily remedied by double buffering, when required. Segments only have access to an active block of data while the DMA process is filling the inactive block. Little overhead is consumed to support either the data area or the pointer management. The only other technique (to ensure data consistency) would be to copy large blocks of information into "local" access areas. This is prohibitive in time since it requires CPU intervention rather than the DMA process. Interrupts during a copy would have to be disabled, which could further degrade performance.

Data consistency for mission computers that are acting as a terminal on the bus is often overlooked. Frequently both the hardware and software do not provide adequate information to detect a DMA process into a given block of memory. Extra time delay and hardware-interface discretes sometimes have to be added to allow adequate control after significant production has passed. In selecting a design, the controller features for the bus are often more heavily weighted than the terminal features; this can contribute to the oversight.

2.3 Time Tagging Data

The F-16 avionics system supports a subsystem need to extrapolate data for integration of sample points over time. For example, the INU provides a method of time tagging the navigation data by use of an interval timer. The time tagging of data is closely associated with particular sensor data. Control is provided through MIL-STD 1553 commands so that such a time-tag can be reset upon the occurrence of a given event.

When the transmissions are ordered within the minor frame, data from various subsystems can be easily correlated. Such capabilities are necessary for supporting Kalman filters as well as tracking subsystems. Thus, the subsystem can then use time-ordered data in its buffers.

Time tagging data greatly simplifies the subsystem algorithms that extrapolate the data. The bus control resets the time tag counter and transmits the data upon a periodic basis. Changes in the state of the system, or transitions, are signaled through appropriate control information included in the message-block mode-word. The subsystem can then transition to the appropriate computational state, e.g., transitioning a subsystem from a scan to a track mode.

2.4 Loosely Coupled Mission Computers

A simple, reliable approach to multi-computer configurations is to maintain loosely coupled functional partitioning. The software execution delays are often incurred when a centralized control policy is allowed to be implemented. In a research project, the F-16 mission software was demonstrated as a loosely coupled, dual computer system. This dual system had the same small executive structure as the basic program. The system control portion utilized the same algorithms, with few changes, to monitor system states and required few table changes. The second computer also contained a backup bus control algorithm; both computers supported the navigation functions. The resultant system did not double the availability of system resources or reliability; therefore, multiple-computer configurations require additional mechanization definition for degraded hardware states and a more complex hand-off of control. The reconfiguration of the system was facilitated by the design principles discussed in the next section.

3. Controllability

One of the important considerations in producing real time software is that the resultant system be controllable. Controllable is defined in terms of Finite State Automata Theory in which the states of the system are finite and the transitions are deterministic, i.e., a finite-state machine. The "a priori" nature of both the inputs, outputs, and task states allows the system to be designed as a finite-state machine.

Three design concepts adopted for the F-16 assisted in achieving the goal of a controllable, predictable system software. These concepts were integral to the design of the system control component, which is responsible for implementing the activities of a finite-state machine. The first concept was that the entire system be implemented with positive-logic; the second concept was to use table-driven software; and, the last concept was to eliminate scheduling anomalies that might arise in executing executive service requests.

3.1 Positive-Logic

In driving the task state transitions, system control implements positive-logic. In other words, a specific input is required in order to accomplish the transition. On completion of the transition, the event status sets are initialized by use of a table look-up strategy. An example is to schedule "air missiles" upon pilot selection of "missile override". The segments of system control do not have to incorporate additional event evaluations in order to "remember" previous states across such transition boundaries. Figure 3-1 shows four possible entry states-A, B, C, and D. A transition is a function of input data. On performing the transition, the system software must generate the appropriate output data to allow the system to operate in the new-state, in the example state F or G. Positive-logic infers that the system software does not have to save the information on how the transition to E occurred. Such an approach provides more orderly state transitions and requires fewer transition-related status-computations.

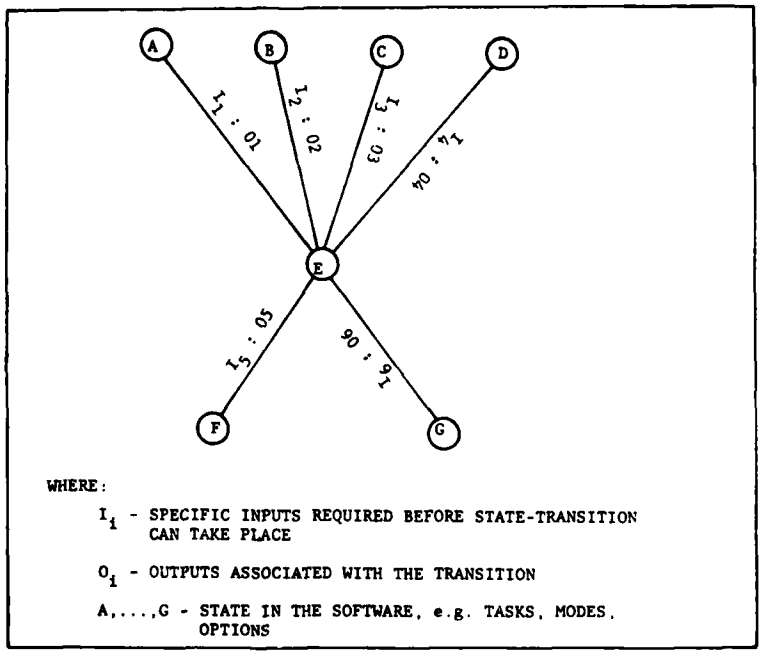


Figure 3-1 State-Transitions are Implemented with Positive-LOGIC

To implement positive-logic requires an early decision in the operational mechanization of the system. An area of the F-16 mission program that did not originally use positive-logic was involved with cursor control and airpoint slewing on the displays over mode and option changes. The original design perspective did not need logic states as the system was to be simple. A decision was made to incorporate positive-logic when reviews discovered the increasing complexity and volatility of the program. For those situations that were unable to be implemented with polisive logic, additional overhead was required to support protection of such information and to deactivate task execution. In such areas, it may be difficult to maintain the software when the tasks are multiprogrammed to achieve this type of affect.

3.2 Table-Driven Software

Many of the segments implement table-driven algorithms. These segments control symbology, scheduling, bus communications, switchology, self-test reporting, and sighting option updates. The advantages of table-driven software are (1) the algorithms are small and remain unaffected by changes, (2) changes are localized to table entries, (3) tables can be more easily reduced than code, and (4) the resultant system is readily testable.

Some of the table-driven algorithms that were implemented in the FCC are well known. Two representative examples are (1) hash-coding for ballistics table look-up and (2) decision tables to select symbology as a function of system mode and option selections. Weapon data and symbology requirements are frequently changed as the fire control mission is revised over the life time of the avionics.

One example of table-driven software is in the system control segment that selects symbology as a function of system mode, switchology, and system states. Figure 3-2 shows the desired symbology for a given situation. Table 3-1 is a portion of the symbology with the associated logic for selecting each item on the display as a function of system mode. This table is derived by the system engineers in conjunction with pilot inputs.

To create a table-driven approach, the program representation of the logic statements are grouped into "cases". For example, the same criteria is used to declutter the flight-path marker in the modes for ALT CAL, FIX, and BASELINE. All other equations that are unique to these three modes are placed in the same case, e.g., altitude and airspeed carets.

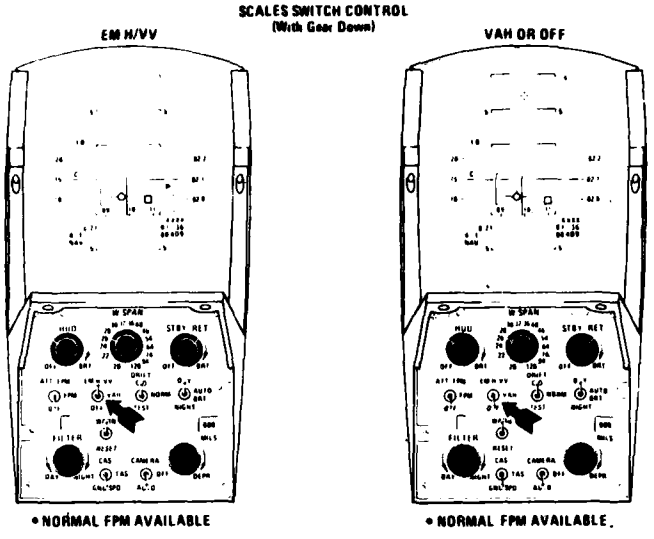


Figure 3-2 Navigation Display Options

SYSTEM MODES	SYMBOL SELECTION **										SYMBOLOGY AND SCALE CONTROL **																								DISPLAY DATA @						CIRCLE CONTROL			
	FPM ON	DIAMOND ON	ADJUSTABLE RADIUS CIRCLE ON	6 MIL CIRCLE	A/G TARGET DESIGNATOR ON	TISL TARGET DESIGNATOR ON	E-O RETICLE ON	BOMB FALL LINE ON	8 MIL CIRCLE ON	FLASH FLIGHT PATH MARKER	MAGNETIC HEADING ON	PITCH LINES ON	BREAKAWAY ON*	FUEL ON	ALTITUDE SCALE ON	AIRSPEED SCALE ON	BORESIGHT CROSS ON	GEAR DOWN	MAGNETIC HEADING DOWN	FCC CONTROL	COMBAT EM SCALES ON	TRACK LINE ON	BACKUP MISSILES ON	ENERGY RATE ON	RANGE RATE CARET ON	MISSILE OVERRIDE	SNAPSHOOT	DOGFIGHT	LCOS	VERT BAR ON	SOLUTION CUE 1	VERT VELOCITY ON	AIRSPEED CARET ON	ALTITUDE CARET ON	ANGLE OF ATTACK ERROR ON	FLASH DIAMOND	RMAX (MAX MISSILE RANGE)	RMIN (MIN MISSILE RANGE)	DISTANCE @1 SEC FLIGHT	DESIRED SPEED	SCALE ELEVATION	SPECIFIC ENERGY	ADJUSTABLE CIRCLE RADIUS	RANGE 'L' ON	RANGE GAP ON	4 MIL TIC ON
E-O WEAPONS	23	1				21	1			10	11	12		4	11	11	1	5		1																0	0	0	0	8	0	0	0	0	0
ALT CAL	24	1			1	21				10	11	12		4	11	11	1	5	1	1									25		26	27	28	29		0	0	0	30	8	0	0	0	0	0
FIX	24	1			1	21				10	11	12		4	11	11	1	5	1	1									25		26	27	28	29		0	0	0	30	8	0	0	0	0	0
BASELINE	24	23			23	21				10	11	12		4	11	11	1	5	1	1									25		26	27	28	29		0	0	0	30	8	0	0	0	0	0

PORTION OF THE LOGIC NOTES:

0 - May be set to any value
1 - Displayed in the current mode

•
•
•

23 - INU good
24 - (¬lgup | ILS on | (att/fpm) | fpm) & INU good
25 - flight path marker on
26 - em-h/vv
27 - (home | range | endurance) & lgup & ¬ILS on
28 - home & lgup & ¬ILS on
29 - (ILS on | ¬lgup) & AOA good
30 - speed = { calibrated airspeed, if cas
true airspeed, if tas
ground speed, if ¬cas & ¬tas

Table 3-1 Part of the Required Outputs to the HUD by Mode

After all of the cases are derived from the symbology table, an array of boolean items, 15 modes by 7 cases, is created. The program tests to see if the case-bit is selected to determine the next case to be processed (see Figure 3-3). Table 3-2 shows the Jovial J3B code for computation of the symbology logic and the case-groups.

The alternative approach to the table-driven algorithm is conditional evaluation of the system states. This results in nested IF-THEN-ELSE statements such as in Figure 3-4. Note that the flowchart has two paths, where p is the number of decision nodes.

The table-driven approach is more testable than that of conditional evaluations. A maximum number of tests, in this instance four, required to exercise all the case-groups is readily identified. On the other hand, many more combinations would be required to assure that all paths have been exercised. The formal test procedures (see example in Table 3-3) assure that not only the proper symbology appears under the switch selection, but also that no spurious symbology appears. Other advantages derived from this table-driven approach are given in Table 3-4.

3.3 Elimination of Scheduling Anomalies

Scheduling anomalies occur when the list orders of the task queue are varied and results in increased total run-time. Four different types of anomalies are generally recognized: (1) changing the order of tasks in a queue, (2) removing some precedence relations, (3) changing the number of processes, and (4) decreasing the run time of some tasks. Steps to eliminate scheduling anomalies were taken in the initial design approach.

Several common approaches have been recognized as unsafe software practices in that the system becomes susceptible to nondeterministic states or anomalies. It is unfortunate that these practices often are used in real-time operation and contribute to system degradation.

One such undesirable practice is to allow wait states to occur. Under this practice, the tasks can post executive requests to be placed upon a wait queue until a given event occurs. This creates additional overhead for the executive to perform the management functions. However, the problems arise because an unpredictable number of tasks will be released into the execution queues. The side effects of allowing wait states can be time-outs or uncoordinated messages. Such techniques are easily replaced by repartitioning the segment so that one segment is scheduled until an event occurs and a new task transition is invoked.

Another similar undesirable practice is to schedule asynchronous tasks on the basis of detailed timing relations of subsystems. Such software is vulnerable to subsystem errors as well as the nondeterministic nature of the task queue. One such requirement is found in the F-16; position update information must be communicated to the inertial subsystem with "wraparound" of data in the form of the updated position data. Special, time-dependent and come-from logic states had to be added to system control and bus control segments to initiate the update.

Complex priority scheduling algorithms were eliminated from the software design. For "a priori" systems such as in a mission computer, this technique is not necessary. A simple precedence relation based upon the data and control flow is sufficient to define the task queue. A weakness of priority-scheduling techniques is that they may not guarantee the execution of each task within a reasonable time interval. Overcoming this deficiency in the priority schemes requires the dynamic "aging" of the priorities which again contributes to nondeterminism, anomalies, and operating system overhead.

3.4 Benefit Assessment

The number of special cases that must be handled add to the documentation, maintenance, and operating system overhead. Such systems are difficult to test and it is difficult to duplicate execution conditions, resulting in less-reliable software. Conscious design decisions eliminated many of these undesirable effects. These design considerations were developed on the basis of research on real-time operating system performance (from analysis found in the literature) as well as evaluation of avionic subsystem performance experienced over four programs at General Dynamics [10]. The resultant software is simple, direct, and testable; its reliability has been demonstrated during three years of flight tests and special demonstration programs. The software has an added feature in being measurable. Quantitative measures of system software supports the growth and maintenance over the system life cycle. This is necessary to assure the balanced utilization of system resources.

4. Lessons Learned

The lessons learned through other related demonstration projects are summarized in Table 4-1. The performance of the F-16 mission computer was demonstrated to be reliable; control concepts were workable. The specializing of the mission responses to a table-driven system control was a significant improvement over general purpose, real-time operating systems that are commercially available. The operational performance of the system is deterministic. Figure 4-1 and Table 4-2 show the performance characteristics over the range of mission-mode selections.

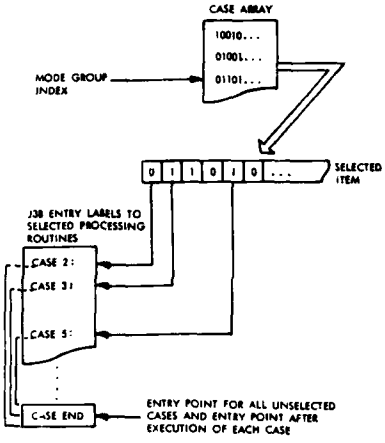


Figure 3-3 Data Structures for the Algorithm and the Particular Computation State

```
    " INITIALIZE CASE B TO CORRECT MASK FOR CURRENT MODE                    "
    CASEB = CASE(DISP.MODE);

    " LOOP THROUGH ALL CASES FOR MODE                                      "
    FOR II = 1 BY 1 WHILE II<10;
      BEGIN % CASE LOOP %

        " DECIDE WHICH CASES SHOULD BE IMPLEMETED FOR THIS MODE            "
        SMASKI = 0;
        IF CASEI < 0;
          SMASKI = II;
        GOTO SMASK(SMASKI);

CASE1:

        " SETS MAXIMUM CIRCLE SCALE AND ADJUSTS CIRCLE RADIUS FOR          "
          EITHER AIM9J OR AIM9L MISSILE.
        MAXCRLSCL = 1.0/3.0;
        IF NOT MSLSKPUNC;
          IF WPNDX = 1;
            ADCLRADU = MAIM9J;
          ELSE
            ADCLRADU = MAIM9L;

        " FLASH ADJUST CIRCLE RADIUS WHEN THE 4 MIL TIC IS FLASHING.       "
        ADRDCIP = TRUE;
        IF FL.MILTIC;
          BEGIN
            ADRDCIP = FL.RET;
            FL.RET = FL.RET XOR TRUE;
          END;

            o
            o
            o

        GOTO CASE.END;

CASE5:

        " DISPLAY FLIGHT PATH MARKER ON THE HUD IF ATT.FPM OR FPM IS       "
          SELECTED ON THE HUD DECLUTTER SWITCH OR LANDING IS TRUE
        FLPTMKON = (LANDING OR TEMP81) ;
        VER.BAR = FLPTMKON;

        " DISPLAY VERTICAL VELOCITY ON THE HUD AS A FUNCTION OF THE        "
          HUD DECLUTTER SWITCH POSITION.
        VEP.VC/ STEM.HD.VV;

        " DISPLAY ALTITUDE CART ON THE HUD IF NOM HAS BEEN SELECTED        "
          AND LANDING HAS NOT BEEN SELECTED.
        ALT.CART = NOM AND NOT LANDING;

        " CHECK FOR VALID CASTAS.                                          "
        IF (CASTAS<3);
          BEGIN
            " DISPLAY AIR CART ON THE HUD IF RANGE OR NOME OR ENDR         "
              HAVE BEEN SELECTED AND LANDING HAS NOT BEEN SELECTED.
            AIR.CART = (RNG.OP.NOM OR ENDR) AND NOT LANDING;

            " SETS DESIRED AIRSPEED TO THE SAME CALIBRATION AS THE AIR     "
              SPEED THE PILOT IS CURRENTLY OBSERVING.
            DESPEED = ADESPEED(CASTAS);
          END;
        ELSE
          AIR.CART = FALSE;

        " DISPLAY ANGLE OF ATTACK ERROR BAR WHEN AOA AND LANDING ARE       "
          TRUE
        AOA.ERR =LANDING AND APHATVAL;

        " TURN ON MISSILE DIAMOND WHEN INU IS GOOD.                        "
        MSLDMOON = INUVALFLG;
        GOTO CASE.END;

CASE6:

        " DISPLAY AZIMUTH STEERING ON HEO WHEN SYSTEM HAS KNOWLEDGE OF

            o
            o
            o

CASE, END: END;
```

Table 3-2 Jovial J73 Code to Implement Table-Driven Processing for HUD Outputs

```
Level          "J3E Code"

  2          IF AIRTOAIR;
  3            BEGIN
  3              IF DGFT;
  4              .  "SYMBOLOGY--OPTION"
  .              .
  .              .
  4              END
  3            ELSE
  4              IF LOCKON;
  5              BEGIN
  .              .
  .              .
  .              .
  5              END
```

Figure 3-4 CONDITIONAL CONTROL FOR SELECTING SYMBOLOGY

Head-Up/HDO Display Function Test

PROCEDURE IDENTIFICATION: HD.P.01.02

CONTRACT END ITEM: FCC OFP CI #14260

PRIMARY FUNCTION: Symbology Control

Test Objectives

1. Verify basic symbology and mode related symbology appearance in all FCS modes. (Subtest 1)

2. Verify conditional displays [illegible] and 15 and HDO [illegible]

Test Configuration

This test procedure requires the OFP to be running [illegible] FCC, and the simulation program to be running [illegible] All DTS control panels must be operational [illegible] HUD must be installed in the DTS. The [illegible] is not required. Configure the [illegible] Standard DTS Setup.

Case 1

o

o

Step 5 HUD window 8 should read NAV.

Check the appearance of the following HUD symbology.

		YES	NO
Flight Path Marker	- off		
Air-to-Ground Target Designator	- on	___	___
Diamond	- off	___	___
TISL Target Designator	- off	___	___
Magnetic Heading Scale	- off	___	___
Pitch Bars	- off	___	___
Altitude Scale	- off	___	___
Airspeed Scale	- off	___	___
Boresight Cross	- on	___	-
Vertical Bar (NAV steering)	- [illegible]		
Vertical Velocity Scale			
Airspeed [illegible]			

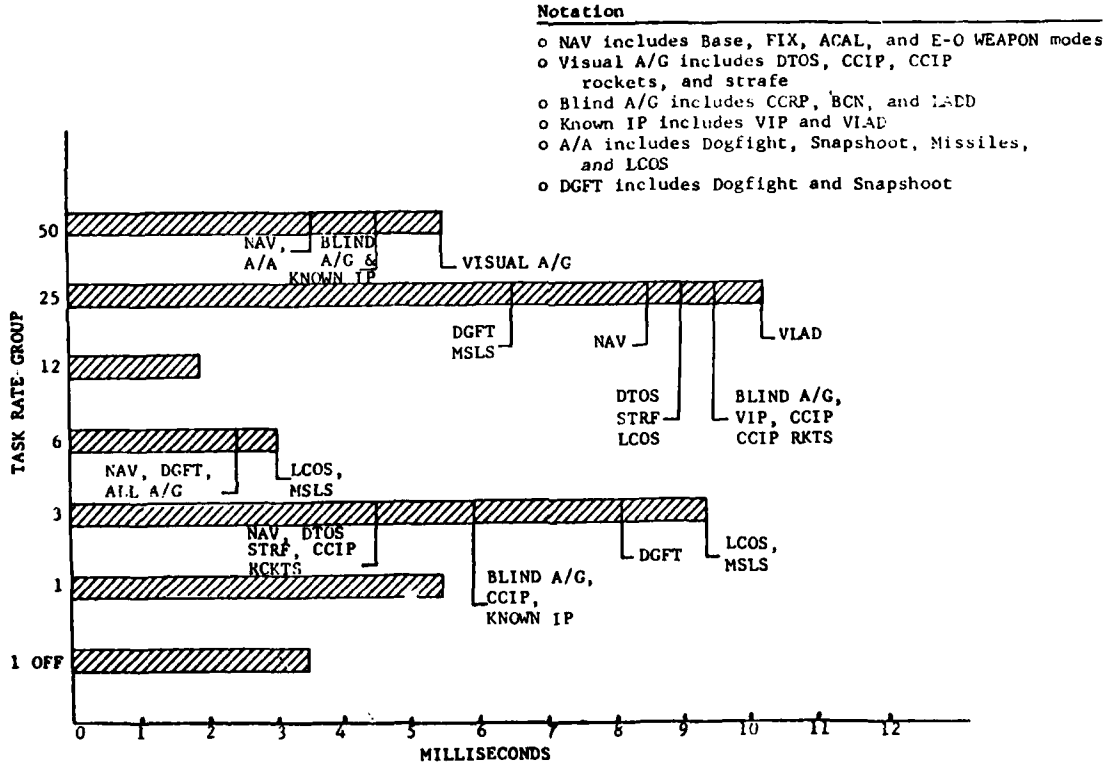
Table 3-3 Sample HUD Test Procedure

- o Common Equations
- o Avoid repeated, nexted IF
- o Rapid access
- o Deterministic, unambiguous
- o Each index/case can be explicitly tested

Table 3-4 Advantages of Table-Driven Software

Positive logic	o Eliminate the "come from" states which reduces the number of combinatorics for the event set.
Loosely-coupled computers	o Minimize the amount of control on communication, scheduling, and state-selection routines.
Asynchronous systems	o Difficult to determine hardware clocking is in step. Must consider impact of operating system and communication overhead delay.
Table-driven control	o Most simple to change & optimizations are easily identified
Data consistency	o Coordination requirement for target sighting, navigation, and aircraft status

Table 4-1 Lessons Learned from Related Projects



TIMESLICE & MAIN LOOP TIMING FACTORS	TYPICAL EXECUTION TIME (MILLISECONDS) AND DUTY CYCLE UTILIZATION											
	NAV-TYPE MODE		VISUAL A/G		BLIND & KNOWN IP A/G		LCOS		MISSILES		DGFT & SNAPSHOOT	
TIMESLICE	MINOR FRAME	MAJOR FRAME	MINOR FRAME	MAJOR FRAME	MINOR FRAME	MAJOR FRAME	MINOR FRAME	MAJOR FRAME	MINOR FRAME	MAJOR FRAME	MINOR FRAME	MAJOR FRAME
50/25	12.2	195.2	14.8	236.8	14.2	227.2	12.7	203.2	10.7	171.2	10.7	171.2
50/12	6.0	48.0	7.8	62.4	6.9	55.2	6.0	48.0	6.0	48.0	6.0	48.0
50/6	6.1	24.4	7.9	31.6	7.0	28.0	6.7	26.8	6.7	26.8	6.1	24.4
50/3	8.2	16.4	10.7	21.4	10.5	21.0	13.1	26.2	13.1	26.2	11.8	23.6
50/1	9.1	9.1	10.9	10.9	10.0	10.0	8.1	8.1	9.1	9.1	9.1	9.1
50/1 OFF	7.2	7.2	9.0	9.0	8.1	8.1	7.2	7.2	7.2	7.2	7.2	7.2
TOTAL PER MAJOR FRAME (640 MS)	NA	300.3	NA	372.1	NA	349.5	NA	319.5	NA	288.5	NA	283.5
TIMESLICE DUTY CYCLE UTILIZATION	NA	46.9%	NA	58.1%	NA	54.6%	NA	49.9%	NA	45.1%	NA	44.3%

MAINLOOP	SOLUTIONS PER SECOND	SOLUTIONS PER SECOND		SOLUTIONS PER SECOND			SOLUTIONS PER SECOND	SOLUTIONS PER SECOND	SOLUTIONS PER SECOND	
		EXCEPT DTOS	DTOS	LADD, VLAD	CCRP, BCN	VIP			DGFT	SNAP SHOOT
TRACK 1	0	11.2	11.2	9.0	12.2	12.0	67.1	9.8	6.0	74.6
TRACK 2	43.5	3.8	1.1	4.0	1.8	1.1	4.6	9.1	9.5	20.5

Table 4-2 FCC Program Execution Timing

Several observations on the adaptability and portability of the algorithms have been noted during the two block updates (see Table 4-3). An algorithm analysis must be performed to determine its suitability for accommodating new requirements. Added resource management also impacts the system design concept. Finally, a method for allocating transactions to a heavily loaded resource must be carefully derived.

A major benefit of these design decisions and conventions adopted for the F-16 operational software was the ease in which the application segments were implemented, integrated, tested, and maintained. In fact, the program was ready for flight testing just eight months after coding began; within 14 months it had successfully passed Formal Qualification Testing, one month ahead of schedule.

System maintenance ease was experienced in the variety of modifications required for the demonstration programs (listed in Table 4-3). In all of these efforts, the executive component was not changed. The system control component was the area that is most impacted by mission-dependent changes. Furthermore, with the table-driven software and the positive logic, three different software engineers assigned at various times in the program to support/maintenance have been able to easily do so with minimal training and effort. At no time did the changes require a major redesign of the system control algorithms; the changes were isolated to the data structures.

Table 4-3 Demonstration Programs

- o PENGUIN study
- o Saber, Atlas, and HMS
- o Aim 7
- o Dual Processor IRAD
- o Two Block Updates
- o Firefly IRAD
- o AFTI Phase I Simulation
- o JTIDS Simulation
- o AMRAAM study

4.1 Growth Versions

Expansions, and consolidations, of the mode- and option-dependent tables were easily made without major modifications to the segment algorithm. Often these features could be demonstrated through a "patched" change to tables for laboratory evaluation before the change was committed to source code changes. Most of the block changes were required for the symbology and switchology management function, which are 80% table-driven.

Inserting and deleting tasks were readily identified by updating the scheduling digraph. Only the system control segment that activates the task has to be changed. This change is minor and incorporates both the decision-logic and task-control statements. The impact to the memory and duty cycle can be easily estimated with reasonable accuracy.

4.2 Algorithm Trade-Offs

One greatly neglected area in real-time software design is the algorithm data structuring. These structures are well-supported by the higher-order language, Jovial. Frequently, memory management schemes with pointer control are needed to minimize the amount of memory required for the functions.

One example for a software/algorithm trade study is in selecting sort routines [8]. For a small number of items, a "bubble-sort" can easily accommodate the tasks. However, for a larger number of points a radix-shell technique may be necessary to improve performance. These algorithms are well-known with adequate supporting analysis as to the benefit of each approach. Other similar algorithm areas that have been well researched are memory management, search techniques, decision-table processing, and the sparse-matrix method. In other words, when algorithms must be modified to accept new functional requirements, the suitability of the approach must be assessed for the amount and type of data utilized in the algorithm.

During a block update, the increased complexity of performing calculations to specific mode-related sighting points was noted; therefore, fixtaking componet analysis was made of the trade-off between function "calls" and data structures. Good savings were realized in reorganizing the method for calculating sighting points from common subroutines and linking a list of the desired sighting points. The algorithm to calculate the range and bearing makes maximum use of the geometry of the situation. The addition of new system modes caused major retasking of fixtaking and indirectly caused new supporting system states to be managed by the system control component. The benefit derived from implementing pointer-tables to select the sighting options was approximately a 200-word code saving. Other derived benefits were those associated with a reduction in the number of paths through the program. The time required to establish parameters for various tasks and subroutines was reduced. Now as new modes are added, the fixtaking component can easily be extended to accommodate new options. The last change did not impact either the computation or the pointer table. It was also noted that system testing was facilitated, since the code path was virtually the same in all modes and only the selected pointers to the desired options had to be monitored. Finally, the resultant documentation and code are more readily understood with the more direct specification of the computational requirements.

4.3 Bus Resource Management

The bus is not an infinite resource. Multiported computer systems are one approach to extending this resource. In systems theory, the most-reliable systems are hierarchical in nature. Therefore, the growth to a multibus configuration is planned to be hierarchical with computers interconnecting more than one bus.

The hardware required to support a DMA capability for multiple buses should have a minimum impact on the CPU. Realtime software is usually required to maintain performance duty cycle range. Constant disruption of the program or inordinate control for port-to-port coordination may have serious overhead penalties that could cause degradation in either the bus or the execution duty cycle.

Another software difficulty with current DMA schemes is to determine what portion on memory is being accessed when the bus is busy. The only safe practice available is for the software to access DMA areas when the bus is not busy. In many systems, the software is unable to make good use of such idle periods.

Interface guidelines [11] were established to improve bus and memory utilization. New subsystems should incorporate these guidelines. Minimizing "special cases" greatly improves the opportunity for utilizing existing algorithms, such as the bus control. Often subsystems are interchangeable. Commonality of the interface data formats and requirements have significant resource savings by allowing more generic algorithms.

4.4 Multiple Processors

In multiple processor configurations, it is essential that the degraded modes appear the same as the primary modes as seen by the operator. To do otherwise, complicates the pilot workload. Repartitioning a mission function into two processors is not the most efficient use of the resource. Rather, the incorporation of these resources should be in support of new mission functional requirements and new subsystems.

5. Summary

The primary design considerations that contributed to the reliable performance of the F-16 mission software were (1) structuring of the operating system, (2) insuring that the software is controllable, and (3) guaranteeing reliable communications. The resultant design is extensible, reliable, and reconfigurable. These features have been demonstrated during ECP changes and research demonstrations. The software is written in 90% Jovial with only those machine-dependent or time-critical tasks in assembler language. This software has accommodated new system modes, subsystems, and algorithms without major changes to the operational environment.

The author would like to acknowledge the assistance of the F-16 software mechanization and program team; in particular, M. E. Cantrell and S. A. Alford. The author also appreciates the review comments of J. D. Engelland, D. E. Sundstrom, J. C. Ruth, and F. Hubans, Jr.

6. Bibliography

[1] Military Standard. Aircraft Internal Time Division Multiplex Data Bus, MIL-STD 1553, 30 August 1973.

[2] Lee, R. Q. and G. England, The Digital Airplane. Astronautics and Aeronautics, Vol 16, No. 1, Jan. 1978, pp. 58-64.

[3] Engelland, J. D. The F-16 Avionic System-Structured for Affordable Performance, Naecon '77.

[4] Scott, C. Missile Intercept Confidence Factor-An Advanced Air-to-Air Missile Launch Envelope Display Concept. Naecon '79, pp. 621-626.

[5] Klos, L. C. An Interface Management Approach to Software Development. Naecon '78, pp. 741-748.

[6] Sundstrom, D. E., W. B. Anderson, and S. A. Alford. F-16 Multiplex: A system perspective. Presented at 2nd AFSC Multiplex Data Bus Conference, 10 October 1978.

[7] Kohavi, Z. Switching and Finite Automata Theory. McGraw-Hill, 1979.

[8] Knuth, D. E. The Art of Computer Programming Vol 1,3. Addison-Wesley, 1976.

[9] Peterson, J. L. Petri Nets. ACM Computing Surveys, Vol 9, No. 3, September 1977, p. 223-252.

[10] Operational Software Concept. General Dynamics, AFAL-TR-75-230, and Softech, Inc., AFAL-TR-77-78, August 1977.

[11] Edwards, J. A. Inside MIL-STD 1553: Interface Format Guidelines. Naecon 1979, p. 419-425.

MAIN COMPUTER SOFTWARE FOR THE MRCA TORNADO

by
K. Sanderson
Procurement Executive, Ministry of Defence
St. Giles Court
St. Giles High Street
London WC2

SUMMARY

Two versions of TORNADO are being produced; one is a UK-only requirement and the other a tri-national requirement. The latter is the subject here since it has a considerable development and production time lead, and the more complex overall management and industrial organisation. This is outlined with particular reference to the role of the national avionic system companies in the participating nations and the organisation provided by them for the design and development of the avionic system, including the software. The specification of the top level avionic design requirements and a description of the computing hardware and data transmission arrangements provide an introduction to the specific software development topics. These cover the specification of the software requirements, the development of the Operational Flight Program, the software structure and documentation, hardware-software integration and testing facilities, documentation for the control and reporting of the testing, configuration control aspects and production software modification control.

1. BACKGROUND

The PANAVIA 200 MRCA TORNADO aircraft has been developed, and is now in production, as a joint venture between the United Kingdom (UK), the Federal Republic of Germany (FRG) and Italy (IT). TORNADO is by far the largest and most complex military aircraft project ever undertaken in Europe, with a production requirement currently foreseen as 809 aircraft of which 385 are for the UK, 324 for FRG and 100 for IT. The targets for national funding and work-sharing are set at UK 42½%, FRG 42½% and IT 15%.

From the outset, operational flexibility was specified as a fundamental design requirement. The resultant aircraft with variable geometry wings has both short field and Mach 2+ performance. A comprehensive digital avionic equipment fit gives it a versatile mission capability; the operational scenario includes:

- Close air to ground support
- Interdiction
- Strike
- Air Defence
- Reconnaissance

The long range Air Defence role is a UK-only requirement. Optimum mission capability in this role demands significant changes in the avionic equipment fit. Therefore, TORNADO is to be produced effectively in two versions. The first will be the originally specified, tri-national, "Interdictor-Strike"(IDS) version, required by all three participating nations. The second will be the UK-only, Air Defence Variant (ADV), which will however retain maximum commonality with TORNADO IDS in respect of all airframe and equipment (including avionic equipment). All differences will be changes essential to the optimisation of the Air Defence mission capability.

In the particular context of the TORNADO computing system, software structure, software test and validation arrangements, documentation, etc, (ie the general subject of this review) there is no essential difference, other than program content, between the TORNADO IDS and ADV versions.

However, TORNADO IDS was conceived first and retains a considerable development time lead, particularly in respect of software development, over TORNADO ADV. Also, TORNADO IDS being truly tri-national in terms of funding and work-sharing, has a considerably more complex (and hence more interesting) overall management and industrial organisation. For these reasons, the following paragraphs are based on TORNADO IDS development practice.

2. OVERALL MANAGEMENT AND INDUSTRIAL ORGANISATION (FIG 1)

With a few exceptions, (eg engine design and development) the design, development and production programme for TORNADO is managed and implemented by Panavia Aircraft GmbH (Panavia) under contract to, and under the general direction of, the NATO MRCA Development and Management Agency. NAMMA is effectively a tri-nationally staffed project office set up by the Ministries of Defence (MODs) of the participating nations to act as their Executive Agency. NAMMA is relatively thinly staffed and hence, during the project definition and development phases, has been heavily supported, guided and advised by the national MOD specialist staffs. For mutual convenience and collaboration, NAMMA and Panavia are co-located in Munich, FRG.

Panavia is a tri-national weapon system management consortium, formed and staffed by the (then) British Aircraft Corporation (now British Aerospace) (BAC/BAe) 42½%, Messerschmitt-Bolkow-Blohm (MBB) 42½% and Aeritalia (AIT) 15%. These partner companies constitute Panavia's first-level sub-contractors for design, development and production requirements. For the present purpose it is sufficient (although a vast over-simplification) to consider Panavia as divided into Procurement and Systems Engineering (Avionic) Monitoring functions. The latter is staffed by MBB giving that company the overall avionic

systems engineering management responsibility.

The Panavia procurement function for the avionic system is exercised generally at policy and co-ordination level, based on tri-national decision in the official NAMMA/Nations environment. The task of contractual interfacing with, and the commercial oversight of, the national avionic equipment suppliers is carried out by the three participating companies BAe, MBB and AIT acting as Panavia's agents. These companies each provide and manage an avionic test rig for full scale system integration of the avionic equipments. However, for avionic system design, the technical specification of avionic equipment requirements and the technical oversight/approval of avionic equipment supplier activity, different arrangements applied.

3. THE ROLE OF THE NATIONAL AVIONIC SYSTEM COMPANIES

At the outset of the TORNADO project and for reasons (largely industrio-historical) which are beyond the scope of this review, avionic system design capability within the participating nations resided largely within specialist national avionic system companies (NASCs). The major NASCs - Easams (UK), ESG (FRG) and SIA (IT) - formed a consortium for early project definition, but this was soon dissolved and the Avionic Development Contract was placed by Panavia on Easams as a first-level sub contractor. As stated in the previous paragraph MBB have, within Panavia, the lead function for avionics and hence the task, on behalf of Panavia, of supervising, monitoring and managing as necessary, Easams detail design and development of the TORNADO avionic system.

The organisation set up by Easams to fulfil this design and development responsibility and to ensure equitable national work-sharing, included both purely national and fully tri-national teams. Sub-contracts were placed by Easams on ESG and SIA (implicitly, also on Easams itself!), to provide three national "In-House Teams" and the staff for two fully tri-national teams - the Central Design and Management Team and the International Software Team.

The Central Design and Management Team. The CDMT is a fully tri-national team, under overall UK management, and co-located with Easams at Camberley, UK. The staff is drawn from Easams (UK), ESG (FRG) and SIA (IT), and so far as is practicable, in the usual project ratios 42½%; 42½%; 15%, respectively.

In brief, the CDMT is responsible for all aspects of the avionic system and sub-systems design, performance and effectiveness, including software. The responsibility includes the system level policy and co-ordination aspects of design factors such as Reliability, Maintainability, Environmental Qualification Test, Electromagnetic Compatibility, Quality Control, Configuration and Interface Control, Human Factors, etc, etc. In addition, the CDMT manage and control the technical programme, planning and analysis of avionic flight trials carried out by the Buccaneer "hack" aircraft, operated on their behalf by BAe.

The International Software Team. The IST is also a fully tri-national team, but under overall FRG management, and co-located with ESG at Munich, FRG. Like the CDMT, the staff is drawn from Easams (UK), ESG (FRG) and SIA (IT), and so far as is practicable in the usual project ratios.

The IST is, apart from its remote location, functionally an integral part of the CDMT. It is responsible for the generation, initial test, documentation and configuration control of the TORNADO main computer software to satisfy the CDMT's system requirements. The software generation task covers three main areas - the Operational Flight Program, the ground engineering test programs and support software for the CDMT controlled (Stage 1 and Stage 2) test rigs.

The "In-House" Teams (IHTs). The CDMT is supported in their tasks by the three IHTs which in effect implement the CDMT's design and policy requirements, generally at national avionic equipment supplier level. The IHTs also provide and manage the CDMT controlled (Stage 1 and Stage 2) test rig facilities which constitute the main software proving and initial avionic system integration facilities.

4. AVIONIC DESIGN REQUIREMENTS (FIG 2)

Avionic system project definition could be considered complete when the participating Nations had agreed a substantially common operational requirement and when all parties - Nations, NAMMA and Panavia - had a reasonably clear picture of the aircraft characteristics and of the avionic equipment fit and functional characteristics, needed to meet the requirement at acceptable cost. At this point in time, circa 1970/71, it became possible to collate preceding design study information, decisions, etc, into a Performance and Design Requirements (PDR) document, which became the formal contractual basis of agreement between NAMMA and Panavia for full development.

For convenience in terms of specification, functional commonality, operational specialisation, etc, the avionic system was divided into a number of sub-systems. In the computing/software context, the most relevant ones are:

- Computing
- Navigation
- Displays and Controls
- Weapon Delivery
- Terrain Following/Automatic Flight Director

For each sub-system a definitive specification was produced reflecting the relevant parts of the PDR in more detailed, functional, engineering and performance terms. Necessarily, these specifications included, either explicitly or implicitly, the top-level functional specification of the software requirements.

Below the definitive Sub-System Specifications, were produced hardware specifications for the individual avionic equipments and also the detailed operational Software Requirements (SWRs) documents. Both reflected the Sub-System Specification requirements in appropriate finer detail.

All of this basic avionic design requirement documentation was subject to iterative review and updating as detailed design and development proceeded. All changes to the primary documents - PDR and Sub-System Specifications - required the approval of NAMMA and the Nations, as did any change to the SWRs that necessitated a corresponding change in a Sub-System Specification. For this purpose, Sub-System Review Meetings between the Nations, NAMMA and Panavia, were held regularly throughout development.

5. THE NAMMA SOFTWARE WORKING GROUP

Subsequent sections deal with the management and control of the translation of the sub-system and software requirements into the developed, tested, deliverable software. To assist NAMMA in its task of monitoring and managing the software development programme on behalf of the Nations, the NAMMA Software Working Group (SWWG) was instituted at an early stage and functioned throughout development. The composition of the SWWG includes representation of all relevant interests on both the official and industrial sides under NAMMA chairmanship. The SWWG is concerned with monitoring for assessment and review in such areas as:

- status of Software Development Schedule.
- progress of development and testing.
- hardware/software interface.
- computer store and time loading.
- problem areas and remedial actions.
- technical aspects of SWRs and associated change requests.
- documentation and configuration control procedures.
- customer documentation of the software.
- development of software production modification assessment procedure.
- inter-nation liaison on in-service software maintenance policy

6. COMPUTING HARDWARE AND DATA TRANSMISSION

The TORNADO computing system architecture may be described as being of the semi-distributed type; however, this term covers a number of possibilities. What is meant here is that in general the individual avionic equipments are interfaced to a central Main Computer (MC) containing the Operational Flight Program (OFP), which constitutes the majority of the overall computational task. In addition, a number of smaller computers perform dedicated functions in various of the avionic equipments and this constitutes some distribution of the overall computational task. Although, there is no redundant computation of MC OFP functions carried out in these "peripheral" computers, this does not imply the loss of all navigation and mission capability in the event of MC failure, since in this event, the avionic system can still function albeit with reduced performance.

The Main Computer (MC), designed and manufactured by Litef, FRG, is usually referred to as the Litef Spirit III. The design is conventional for the early 1970s as is the circuit implementation in Bipolar TTL MSI. Thus, the data format is parallel, binary, 2's complement, 16 bits single-length, 32 bits double-length; arithmetic operations are integer, fixed-point and include hardware Multiply (single and double-length) and Divide (single length). Fiftyeight instructions provide for Load, Store, Arithmetic, Compare, Transfer, Shift and Input/Output operations, of which the Load, Store, Arithmetic and Shift groups include double-length operations. The arithmetic/logic unit provides for two independent program levels, each of which has 4 x 16 bit hardware Accumulator registers and 4 x 16 bit hardware Index registers. Direct addressing is up to 128 words/page, 512 words total. Maximum addressing capability is 64K words (K = 1024). The arithmetic/logic unit operation is controlled and sequenced by micro-program.

The memory capacity is 32K words, each of 18 bits (16 data bits + memory protect and parity). The cycle time is 1.5 micro-seconds and the access time 0.45 micro-seconds. Thus, typical instruction execution times are Add/Literal 1.5, Add/Subtract Single 2.0/3.0, Add/Subtract Double 3.5/4.5, Multiply 11.0/11.5, Divide 13.0/13.5 and Transfers 2.0, the lower values being for register to register operation. (A modification is currently being developed to increase the memory capacity to 64K words within the existing volume and to provide a modest (15%) speed increase.)

The Input/Output interface provides 16 (24) Serial Input channels, 18 (22) Serial Output channels, 8 (16) Input Discrete Signal lines and 8 (16) Output Discrete Signal lines. (The brackets indicate the maximum design capacity.) All channels/lines are connected via program controlled multi-plexing hardware to the Direct Memory Access port. In addition, a Special Serial Input Interface is provided for the purpose of loading program and mission data from magnetic tape cassette via the digital replay feature of the TORNADO Cockpit Voice Recorder. Also, included is provision for the input of six independent external interrupts which are part of the 16 level priority program interrupt system.

The complete computer including Memory, Input/Output Interface, Power Supply and Built-In-Test-Equipment (BITE) is contained in a standard 1½ ATR-Short case.

Data transmission between avionic equipments is generally, (ie wherever practical and economic) via standard, serial digital, uni-directional, dedicated transmission links. These comprise 2 pairs of wires carrying 64 KHZ clock on one pair and 32 bit data words on the other pair, giving a transmission rate of 2000 words/second. The 32 bit data word includes 16 bits for data and 5 bits for the data identifier. Thus for the specified sequential and continuous transmission, the data "refresh" rate varies between 62.5 and 2000 times per second, depending on the number of data items (1 to 32) carried on the channel. The remaining bits of the data word are status bit, parity bit, 3 "spare" bits and 6 synchronisation bits. Together, these allow comprehensive data validity checks, which are implemented in a defined manner.

Equipments not specially developed for TORNADO and equipments such as switch/indicator panels, synchro or analogue controls/displays, etc, are connected to the data transmission network via two Interface Units which provide the necessary signal conversion to and from the standard serial digital format and the necessary standard serial digital interfaces.

7. SPECIFICATION OF SOFTWARE REQUIREMENTS (SWRs)

SWRs reflect the functional and operational requirements of the Sub-System Specifications in detailed software engineering terms. Included are descriptions of the relevant parts of the associated sub-system(s), relevant interfaces, software tasks and crew procedures, together with details of the logic and equation development. In general, the Navigation, Displays and Controls, Weapon Delivery, etc, groups of SWRs reflect the corresponding sub-system functions; however, some overlap is inevitable, eg to avoid duplicating shared functions and data. The following examples illustrate the degree of sub-system breakdown at the SWR level of specification:

Navigation Sub-System

- Navigation Moding
- Present Position Calculations
- Kalman Filter
- Vertical Channel Calculations
- Navigation Calculations for Terrain Following
- Navigation Fixing
- Track Steering

Displays and Controls Sub-System

- Combined Radar/Projected Map Display (CRPMD) and Repeater Projected Map Display (RPMD) - Drive
- TV/Tabular Display and Multi-Function Keyboard (MFK)
- Head Up Display (HUD) - Drive
- Display Recorders
- Co-ordinate Transformation
- Rapid Data Entry

SWR specification is a part of the System Design function of the CDMT. Modelling support at SWR, sub-system and system level is a part of the System Performance function of the CDMT. In particular, software models based on SWRs as source documents enable the contents of each SWR to be checked at an early stage for logic integrity and dynamic accuracy. SWR models are also integrated into larger functional blocks and stimulated from software models of the avionic equipments and the aircraft characteristics, to provide estimates of the overall Navigation and Weapon Delivery, etc, Sub-System performance, and to assess compatibility between functional groups. The results are used later for comparison with the test results obtained from the actual Operational Flight Program software and also for the assessment of the effect of design changes. SWRs are also produced for the various Engineering Ground Test and Integration programs.

8. DEVELOPMENT OF THE OPERATIONAL FLIGHT PROGRAM (OFP)

A progressively staged development was adopted for the production and testing of the OFP, in order to match the staged ground and flight test programme, the phased specification of sub-system and software requirements, variable lead times for the development of the various avionic equipments and for the specification and phased build of the avionic test and integration rigs. Initially, five versions of the OFP (Software Series 1 to Software Series 5) of increasing complexity were scheduled. In the event, it became expedient to combine SS3 and SS4 and to introduce an SS6 version so that the difference between SS3/4 and the complete OFP requirements could be developed and tested in two incremental steps.

The design requirement for each Software Series is defined in the corresponding Design Data Set produced by the CDMT. This specifies all applicable documents and the applicable Issue Numbers. For SWRs, in particular, all applicable changes not yet incorporated in the current issue of the SWRs are identified. Also, any software variations required by particular aircraft, or avionic test rigs, due to avionic hardware differences, are specified in detail.

The following indicates the staged approach to the full OFP requirements.

SS1 - primarily, this provided the special display and recording facilities required for in-flight assessment of the avionic system. These provided for the display of selected MC data on the TV/Tabular display, data recording on magnetic tape for subsequent analysis and Decca Navigator System co-ordinate conversion to Lat/Long. (The Decca Navigator System provides area coverage position fixing, used in this instance as an "external" position reference.) In addition, SS1 contained some of the basic OFP navigation computation, required particularly in relation to the early task of basic navigation sensor performance checking by cross-comparison and against the Decca Navigator System.

SS2 - comprised the Flight Trials facilities of SS1 plus the basic OFP navigation and steering functions, together with the associated Projected Map Display, TV/Tabular Display and Multi-Function Keyboard functions.

SS3/4 - comprised SS2 functions, plus Kalman Filter correction and position fixing additions to the navigation system, additions/extensions of the display facilities, and basic weapon aiming functions.

SS5/6 - comprised SS3/4 plus additions/extensions, particularly in the weapon aiming area, to provide the full OFP requirements, within limits imposed by computer store and time loading. The distribution of new/improved functions between SS5 and SS6 was determined by rig and flight test programme priorities.

At the time when the major decisions on the Computing Sub-System hardware and on the general software development strategy had to be taken, High Level Languages (HLL) for real-time system programming were not in general use as they are today. A HLL was not available for the Litef Spirit III computer. Such languages or their (few) implementations were generally very inefficient in the use of computer storage and time (typically + 30 to 50%). Also, computer store suitable for airborne application was extremely expensive, particularly when considered against the estimated store requirement (then 24K words) and the large number of aircraft envisaged. The decisions were made therefore to specify 32K words of store (leaving some spare for in-service development) and to write the Operational Flight Program in the Litef Spirit III assembler language which was already available. These decisions were eventually fully justified, but even so and almost inevitably, the 32K word store became fully allocated as did most of the computing time at peak demand periods, with some desirable program functions having to be omitted and with no spare for in-service development.

Fortuitously, some 5 years after the original decisions, with the advances that had been made in large scale integration and electronic packaging techniques generally, it became possible to pack 64K words into the original 32K word space, for approximately the same price and power consumption. This modification is currently being developed, together with further modification to provide a modest (15%) speed increase. The 64K words of store, with its associated new program - Software Series 7 - will allow both the inclusion of all the outstanding requirements which had to be omitted, and the reinstatement of all the desirable program functions which had to be deleted, in order to contain the SS6 program within 32K words. In addition there will be a very generous provision of spare store for future in-service developments.

9. SOFTWARE STRUCTURE

Program. Each of the on-aircraft MC programs, ie the Operational Flight Program (OFP) and the Ground Test Programs, are complete, self-contained software structures, needing no other software to control or determine their operation. Each program comprises a number of Sub-Systems as illustrated in Fig 3 for the case of the OFP.

Sub-System. A sub-system is an obvious operational and/or functional sub-division of the program software, eg Navigation, Displays, Weapon Arming, etc. In addition to these application oriented sub-systems, all programs include a Supervisor Sub-System and a Common Sub-System. Each sub-system comprises a number of software packages.

Package. A Package implements a specific sub-system function, eg the Moding and Present Position Package of the Navigation Sub-System. A package is also the basic relocatable program unit for program assembly. A package is basically sub-divided into a number of Tasks but, for programming convenience, program efficiency, etc, may also include Routines and Sub-Routines.

Task. A Task is a further functional sub-division of a package. This is normally necessary because different sub-functions within a package will generally be required to be activated at differing iteration rates (Fig 4) and also, because not all of the sub-functions of a package may need to be activated at any particular period, depending on the particular phase within a mission and the operational mode selected. These aspects are considered further with the Supervisor Sub-System below. Tasks may comprise or include Routines and Sub-Routines.

There are two types of task - Base Level Tasks and Freeze Tasks. Base level tasks are the normal application software modules. Freeze tasks have special timing priority over base level tasks, to enable a few low priority packages/tasks to have time critical data at precise iteration rates.

Routine. A Routine is simply any convenient software sub-division of a package or task. The Interrupt Routine is considered with the Supervisor Sub-System below.

Sub-Routine. The Sub-Routine format is aimed primarily at realisation of store economy. It implements some general, frequently required, function and when called from any particular point in the program, returns control to that point on completion. There are four types of sub-routine, Package, General Common, Special Common and Interrupt.

The Package Sub-Routine is a functional part of some particular package; it is exclusive to, located within and may be called by any of the component tasks, routines, etc, of the particular package. (See also Control, Communication and Access Restrictions below.)

The General Common Sub-Routines provide the common mathematical functions for general use throughout the program. They are collected together in a dedicated package within the Common Sub-System.

The Special Common Sub-Routine is shared by, but is exclusive to, a number of packages which have some functional inter-relationship, eg the TV/Tabular Display sub-routines.

The Interrupt Sub-Routine is some general function called from one of the interrupt routines. It retains the same priority level as the calling routine.

Control, Communication and Access Restrictions. Tasks may only be entered from, and on completion must return control to, the Task Scheduler in the Supervisor Sub-System. In particular, tasks (and packages) may not pass control to other packages and all inter-communication must be by means of data parameters, generally located in the data packages of the Common Sub-System. Access to sub-routines and data may be restricted to a particular routine, task or package. This is achieved by location (definition) of the sub-routines and/or data within the respective routine, task or package boundaries. (For structural consistency, it has been proposed that access restriction be imposed also at sub-system boundaries.)

The Supervisor Sub-System. The main functions of the Supervisor Sub-System are to perform hardware/software initialisation, to handle interrupts and to schedule and monitor the running of the program tasks.

The MC hardware provides 16 levels of priority program interrupt. Level 0 has the highest priority and Level 15 the lowest priority. Normal interrupt level (program base level) is Level 16. Each interrupt level has an associated, dedicated, Interrupt Routine to which control is passed by the hardware to service the interrupt, following which control is returned to the interrupted point in the program. The hardware control is exercised via Interrupt Vector and Entry Point Tables in the supervisor software. The majority of the interrupt routines are associated with the various Fault, Input Power, End of Direct Memory Access I/O and External conditions Interrupts. Of particular interest here are the Real Time Clock (Level 4) and the Task Scheduler (Level 15) interrupt routines. The former is activated via the interrupt hardware every 20 mS and in turn activates the Task Scheduler every 20 mS.

The function of the Task Scheduler is to provide real-time control of the sequencing of program tasks according to their assigned iteration rates and priority. The scheduler uses a control table to determine the tasks to be activated during any 20 mS iteration cycle. Not every task is performed in every cycle, the frequency at which a task is performed being its iteration rate. The standard iteration rates are 50, 25, 10, 5 and 0.1 HZ. In general, high iteration rate implies high priority. Freeze tasks due in the current cycle are activated first, followed by base level 50 HZ tasks, followed by base level 25 HZ tasks, etc, etc. At a given iteration rate tasks are activated in their order of appearance in the program. Thus priority is determined by type of task, iteration rate and order of appearance in the program.

The scheduling system is also provided with a limited capability effectively to vary task priorities during different phases of a mission, by selective activation of tasks according to the state of a "permit indicator" at the head of each task, which can be set or reset by any other task.

As necessary, a lower iteration rate task will be interrupted when activation of a higher iteration rate task becomes due. Therefore, in the extreme situation, a lower iteration rate task may become due for entry whilst still active from some previous cycle. This is, in effect, a time overload situation. In this case, the new activation is suppressed as is the activation of any succeeding tasks at the same and lower iteration rates. However, as the more important tasks are all at higher iteration rates and therefore have the highest priority, they will not miss cycles of iteration even at peak loading.

Time permitting, the last task to be entered is the Self Check Facility - the Background Task - which runs until the end of the 20 mS period when it is interrupted by the Real Time Clock routine for the start of a new iteration cycle.

The Common Sub-System. Program facilities available for global (common) access are grouped together in a Common Sub-System. This comprises the General Common Sub-Routine Package and the data packages of the common data base, which provide the majority of the data (non code) storage used by the program. The Global Data Base Package provides the storage for most of the inter-package communication parameters. The Mission Data Store Package provides the storage for the present mission data, eg route, turning points, planned speeds, destinations, fix-points, etc. The Work Store Package provides the temporary (current cycle only) work space allocated to a task when activated by the Task Scheduler. The work store is divided into several areas, each shared by the group of tasks operating at a particular iteration rate.

Software Documentation (Technical Description). UK experience from previous projects, eg Jaguar, indicated the vital need for definitive specification of software documentation (technical description) requirements, to enable the customer to understand the operation of the software, to maintain it, and if so required, to participate with industry in any further development in-service. A definitive specification had been applied to UK air projects from 1974 with the further requirement that the development phase documentation should, whenever possible, be designed to provide the basis for the customer documentation in order to minimise cost and duplication of resources. This system helps to minimise the effects of high programming staff turn over rate, individual idiosyncracies and generally helps to instil a disciplined approach during development. A tri-national definitive specification of software documentation requirements was agreed for TORNADO based on an adaptation of the UK system, which requires four levels of technical description of increasing detail. For TORNADO, the levels adopted are program, package, package components, plus detailed code listings and descriptions, under the generic title of Computer Program Manuals (CPM).

10. INTEGRATION AND TESTING FACILITIES

To cater for the integration and testing of the avionic system (including the software), a number of test rig facilities were commissioned to provide increasingly representative (and therefore increasingly complex) environments in which to exercise the hardware and software. These range from Stage 1 - the software generation and test facilities at the IST - through to Stage 5 - fully representative flight trials in the TORNADO aircraft (Fig 5). Each stage is designed to yield increased confidence over the preceding stage within the limitations of each test rig environment.

Stage 1. This is the working environment of the IST in which the programmers generate, develop and test the TORNADO MC software. The Stage 1 facility is divided into two distinct functional entities - Stage 1A and Stage 1B.

Stage 1A. This facility principally provides for the initial assembly, editing, de-bugging and linkage of MC software modules. It is based on the Host Computer concept using the large Siemens computer at ESG. The main items of host software are the Assembler and the Emulator, both originally provided by Litef as card-based software, but since then, extensively adapted and improved by the IST and now disk-based. The Host Assembler provides conventional assembly facilities - creation of source code files, conversion of source to object code files, syntax checking, error reporting, listing, including cross reference listing, editing, linking etc. The Emulator provides a software model of the Litef Spirit III MC instruction repertoire in which new software can be run, checked for store and time requirement etc. Additional facilities such as object module linking and auto-load paper tape output are provided, making the Emulator self contained.

Stage 1B. This facility provides an environment in which MC software can be exercised, edited, de-bugged and the real time aspects checked in a development model TORNADO MC. The main items of hardware are the MC, with Manual Control Unit and Paper Tape Station, together with a development model TORNADO TV/Tabular Display and Multi-Function Keyboard. A MC Operating System, developed by the IST, provides additional facilities to those available from the standard Litef MC utility programs. The Operating System provides for static program testing and to a limited extent, dynamic open loop testing using test data from paper tape. The Operating System is sufficiently small (some 3K to 5K words depending on choice of facilities) to allow co-residence with the majority of the full OFP software.

This basic avionic rig is connected to and supported by a PDP-11 computer which can simulate appropriate parts of the aircraft avionic environment (open loop) and can dynamically stimulate and monitor programs running in the MC. Dynamic stimulation is based on data derived from the CDMT avionic models and SWR models, resident in this external computer. The results from the MC software under test can then be compared with the expected (modelled) results.

Stage 2. This is an engineering facility designed for:

- primary avionic equipment, sub-system and system integration using the first available development model equipments (electrically representative).
- integration and proving of the software with electrically representative avionic hardware.
- system performance assessment.

Because of the envisaged high workload in relation to the required timescales, two Stage 2 rigs were commissioned, one at Easams UK and one at ESG FRG. These were operated in parallel and generally, at any given time, each concentrated on a different avionic sub-system area. The rigs were provided and operated by the Easams and ESG In-House Teams, to meet the system development work programme requirements of the CDMT. A considerable amount of the Special to Type Test Equipment (STTE) associated with the individual avionic equipments was provided and, for the sensor equipments, in addition to conventional test functions, this provides for open-loop stimulation of the sensor equipments during rig operation. The avionic equipment is augmented by an "external" computer facility, comprising a PDP-11 digital computer and an associated analogue computer. This facility provides further stimulation, as well as simulation and data recording/display functions.

The avionic equipments were integrated into the Stage 2 rigs in a planned sequence of building blocks, depending on their availability and the particular Software Series with which they were required to interface. Hardware integration test procedures ensured compatibility of equipments as parts of a sub-system and integration of the corresponding software functions then enabled software proving and finally sub-system/system functional testing.

Software proving concentrates on testing the software against a very detailed test procedure produced by the rig staff and based on the CDMT Software Test Specification for the particular Software Series. Fault finding and rectification work, on both the software and the test procedure, typically require some 3-6 months per Software Series, after which the software is delivered to the later test Stages 3 and 4. The final issue of a Software Series from Stage 2 is associated with a formal demonstration of the standard of the software against the test procedure, under Quality Assurance supervision. This provides the required evidence that the software is tested sufficiently at a Stage 2 rig to perform adequately at the later test stages.

System Functional Testing is not aimed primarily at the software but is designed to ensure, as far as possible, that integrated hardware and software sub-systems perform in the desired manner and interface correctly to other sub-systems. Typically, a stimulation schedule representative of, eg a typical flight profile, may be output in real-time by the external computer facility and selected MC and system data parameters recorded by the external computer facility for subsequent analysis. The stimulation data file may be derived from the results of a previous test, from in-flight recordings, etc, or from the outputs of aircraft, equipment and sub-system/system models for the specified flight profile and timetable. The "operating system" enabling such stimulation, data selection and recording, etc, functions to be set up, executed (open or closed loop) and replayed for analysis, etc, is the Integration Software written by the IST.

Integration Software facilities include:

- Test definition and set-up
- On-Line Facilities
  - . avionic equipment simulation.
  - . aircraft simulation (lateral and longitudinal) using pre-defined flight conditions. Formerly this was implemented by the analogue computer, latterly by a digital model in the PDP-11.
  - . stimulation (data files/mathematical functions).
  - . display and recording.
- Off-Line Facilities
  - . replay.
  - . comparison of recorded files with files produced from other tests or from software models.
  - . analysis of results.
  - . plotting of results.

Stage 3. This is a CDMT controlled experimental avionic flight trials facility based on two Buccaneer aircraft, which have high performance representative of TORNADO in sub-sonic flight. The aircraft are fitted with second prototype ('B' Model) versions of the major avionic equipments together with the Decca Navigator System for "external" position reference and comprehensive instrumentation for recording avionic performance.

The initial aim was to establish at the earliest opportunity, the in-flight performance of integrated avionic sub-system functions for Navigation, Terrain-Following and Weapon Delivery, in the critical high

speed, low-level, sub-sonic flight regimes, prior to flight in the TORNADO prototypes. This general philosophy has since been continued, anticipating the various phases of the TORNADO avionic flight test programme and providing significant feedback to the software development task.

The aircraft are supported by a limited ground rig providing for maintenance of the aircraft avionic systems and for integration checks of hardware and software. A Ground Replay and Analysis System provides for local monitoring of the test results and collates and formats the data for detailed performance analysis by the CDMT.

Stage 4. Each of the aircraft manufacturers (BAe, MBB and AIT) operate a Stage 4 rig facility which is effectively a ground mock-up of the complete aircraft avionic system, fitted with second and third prototype (B/C Model) versions of the avionic equipments, to provide a system environment representative of the aircraft. The primary tasks are:

- integration and performance testing of the complete avionics system, hardware and software.
- avionic system integration with other aircraft systems.
- flight line avionic support.

The MBB and AIT rigs are closely similar in design based on simplified cockpits with equipments plus wiring in aircraft representative positions and using PDP-11 external computers. However, the BAe rig is an open-plan bench type layout and uses PDP-8 computers. Thus, the integration and operating software also differs between the two rig versions, although both are referred to as Data Acquisition and Simulation Systems (DASS). The MBB DASS is the more complex of the two systems and has a similar capability, in general functional terms, to the Integration Software used at Stage 2.

Software test activity includes Software Proving as part of the System Integration and Performance Testing. Apart from the greater emphasis on integration with the hardware and the more representative environment, the software proving activity follows similar practice to Stage 2 testing. A notable additional requirement is Flight Clearance certification of the software. During the life of each Software Series, the increasing number of amendments and experimental changes to the software (some of which are hardware dependent), the differing avionic build standards of the various prototype aircraft and the progression of the flight test tasks, combine to necessitate the definition of a number of differing configurations of each Software Series for flight clearance certification purposes, which is therefore a progressive task.

Stage 5. This stage comprises the aircraft manufacturers flight test facility using specially instrumented TORNADO prototype aircraft, which are supported by ground replay and analysis facilities and the Stage 4 rigs.

Official Test Centre (OTC) Facilities. An extension of the manufacturer's Stage 4 and Stage 5 facilities is provided at each of the National OTCs. (In the UK this is the Aircraft and Armament Experimental Establishment at Boscombe Down.) Pre-production aircraft are allocated for the customer's official evaluation, but to a significant extent the flight trials are joint contractor/customer trials in the interest of maximum utilisation and efficiency. The aircraft are supported by a Stage 4 type rig at each OTC.

11. CONTROL AND REPORTING OF SOFTWARE TESTING (FIG 6)

In order to define and control the testing of the software on the various rigs, a documentation system is necessary to specify the build standard of the software, the tests to be carried out and to define the test methods according to the particular rig(s) on which the testing is to be performed. The fundamental reference is the Software Test Specification, produced by the CDMT for each Software Series, which defines the tests to be performed largely without reference to particular test facilities. Based on the Software Test Specification, detailed Software Test Procedures are produced for each rig and these tailor the requirements of the Software Test Specification to the test facilities available at the particular rig. Both specification and procedure are divided into definitive test areas, eg by sub-system, SWR, test type; each division is ordered so that the test sequence progresses from simple/general to complex/particular. The procedures are designed to provide step by step test sequences, giving repeatable results.

It is to be noted that the Software Test Specification is produced by the same system design and performance groups within the CDMT which produce the SWRs against which the software is written. Likewise the Software Test Procedures are produced largely by the test rig personnel. In each case, therefore, the influence of the IST programmers is minimised and this provides an important safeguard against any misinterpretation of the designers intentions being carried forward into the functional test criteria.

Although, no formal certification is required by the later test stages when a software series is delivered from the Stage 2 rigs, a baseline record is required of the software standard against the test specification and test procedure, and of any deviations, limitations, untested areas, concessions, etc. For this purpose a formal demonstration against the test procedure is carried out under Quality Assurance supervision. However, both the software and the test procedure will have been subjected to a series of updates during the testing at Stage 2. Therefore for formal demonstration and record purposes, a Test Data Set is produced by the CDMT for each Software Series, which defines both the final software and all applicable sections and Issue status of the final Software Test Specification and Test Procedure.

On completion of the demonstration, a list of deviations against the test procedure is identified - effectively a summary of the subsequent QA Inspection Report. On the basis of this, the CDMT as design authority, produce a Program Acceptability Statement supported as applicable by lists of program deviations, any untested areas, concessions, etc and an indication of any limitations that should be applied pending corrective action. These accompany the software to the later test stages. In slower time, a formal CDMT Test Report is produced based on detailed analysis of the QA Inspection Report.

12. CONFIGURATION CONTROL ASPECTS

Inevitably, during the course of development, many changes to the software were found to be necessary and for the usual variety of reasons. The assessment, co-ordination and implementation of these changes was complicated by the larger than usual number and spatial separation of the eleven separate organisations concerned with the design, development and proving of the software. However, the principles involved were relatively simple and for configuration control purposes, a change could be placed in one of three categories:

- associated with a change to the relevant SWR.
- associated with a hardware change.
- not associated with either a hardware or SWR change.

Furthermore, external to the CDMT/IST, changes could be initiated only in response to a Software Query (SWQ), a Software Requirement Change Request (SWRCR), or a Program Change Request (PCR). The SWQ is, in essence, a brief symptomatic report from a test site of some sort of software defect and a request for CDMT/IST investigation and comment/cure. If a software change is found to be necessary then SWRCR and/or PCR action is followed.

Software Requirement Change Requests. All changes to the SWR documents are first defined in a SWRCR. These may be initiated by NAMMA resulting from changes to hardware, operational requirements, cockpit functions, etc, by the CDMT or the aircraft companies resulting from further definition of the system or performance, incident and defect reports, etc, or by the IST for minimisation of the software, clarification, further definition, correction, etc.

For assessment and approval purposes, it is necessary to distinguish between those SWRCRs having operational implications or hardware change associations (Category 1) and all others (Category 2). Category 1 implies that the SWRCR, if implemented, would reflect a corresponding change into a higher level system or hardware document, ie PDR, relevant Sub-System Specification or Equipment Specification and hence that NAMMA approval is required.

All SWRCRs are assessed by the Software Co-ordination Group within the CDMT and subsequently by the IST and the aircraft companies prior to approval for implementation and inclusion in the SWRs. However, implementation of SWRCRs with hardware change implications must await NAMMA/Panavia negotiation of the commercial aspects of the hardware change. On the other hand, an urgent priority may be assigned for immediate implementation to avoid nugatory work, to match rig test schedules, etc. In any case, independent of the implementation of a SWRCR as a software program change, its incorporation into the SWR is done on a block update basis. It follows that the Design Data Set definition of a Software Series is in terms of SWRs at a particular issue status, plus a list of agreed SWRCRs. The latter list, being subject to continuous negotiation for reasons of relative priority, expediency, work schedules, etc, implies a significant configuration control problem both for the software changes and for the associated documentation, eg the software test specifications and procedures.

Program Change Requests. In principle, any change (not initiated by SWRCR action) to a Software Series delivered from the IST is initiated by a PCR raised by one of the test sites. In practice, this applies to any change to a Software Series delivered from the Stage 2 rigs, since prior to such delivery sufficient direct control can be exercised between CDMT, IST and Stage 2. The IST is responsible for co-ordinating the PCR Control Procedure. A PCR may result in a SWRCR if the SWR is in error or deficient; in this case the SWRCR procedure is followed. In all other cases, including those associated with hardware changes, the IST assess the PCR for acceptability and, if agreed, design and issue a software change and documentation, including hardware-software compatibility aspects, to all test sites.

In order to maintain work schedules at the test sites, it was found to be necessary to establish IST representation at the test sites and to institute a rapid reaction Experimental Change Procedure. Experimental Change Requests are normally initiated by the test site engineers and whenever possible the on-site IST representative, after liaison with the IST, produces a program change for local test and evaluation. A PCR is then raised for normal co-ordination by the CDMT or IST through to formal inclusion in the program as a permanent change, or noted as a special to purpose requirement.

13. PRODUCTION SOFTWARE MODIFICATION

The transition from the development phase to the production phase required the negotiation and agreement of a Production Modification Procedure (PMP) for the authorisation and control of all modification work with respect to Baseline Build Standards (BBS) to be agreed for the complete aircraft and each of its constituent equipments, etc, including MC software. The PMP is operated via the NAMMA Production Modification Control Board (PMCB). Based on recommendations from the Software Working Group it was agreed that all MC software changes with respect to an agreed software BBS would be treated as modifications requiring PMCB approval. Furthermore, it was accepted that a Software Change Committee (SWCC) was required, to monitor software modification work and to advise the PMCB on all aspects of software modifications. The make-up of the SWCC is similar to that of the SWWG but is somewhat smaller in numbers and with rather more emphasis on customer operational and engineering representation. The first tasks of the SWCC were to establish the required standard of the software for first production batch aircraft, within the constraints of the 32K word main computer, and to establish an agreed basis for definition of the software BBS. The initial BBS is now defined in terms of "frozen" SWRs and CPMs corresponding exactly to the software standard agreed for the first production batch aircraft. Hereafter, all software changes reflecting corresponding changes into the "frozen" SWRs and/or CPMs will be treated as software modifications and will be defined in terms of the changes required to the SWRs and/or CPMs.

LIST OF ABBREVIATIONS

ADV	Air Defence Variant
AIT	Aeritalia
BAC	British Aircraft Corporation (Now BAe)
BAe	British Aerospace
BBS	Baseline Build Standard
CDMT	Central Design and Management Team
CPM	Computer Program Manuals
DASS	Data Acquisition and Simulation System
ESG	Electronic-System-GmbH
FRG	Federal Republic of Germany
HZ	Cycles per second
HLL	High Level Language
IDS	Interdicter-Strike
IHT	In-House Team
IST	International Software Team
IT	Italy
K	1024
KHZ	Kilocycles per second
MBB	Messerschmitt-Bolkow-Blohm GmbH
MC	Main Computer
MOD	Ministry of Defence
MRCA	Multi-Role Combat Aircraft
MSI	Medium Scale Integration
NAMMA	Nato MRCA Development and Management Agency
NASC	National Avionic System Company
OFP	Operational Flight Program
OTC	Official Test Centre
PCR	Program Change Request
PDR	Performance and Design Requirements
PMCB	Production Modification Control Board
PMP	Production Modification Procedure
QA	Quality Assurance
SIA	Societa Italiana Avionica SPA
SS1-6	Software Series 1 to 6
STTE	Special to Type Test Equipment
SWCC	Software Change Committee
SWQ	Software Query
SWR	Software Requirement
SWRCR	Software Requirement Change Request
SWWG	Software Working Group
TTL	Transistor-Transistor Logic
UK	United Kingdom

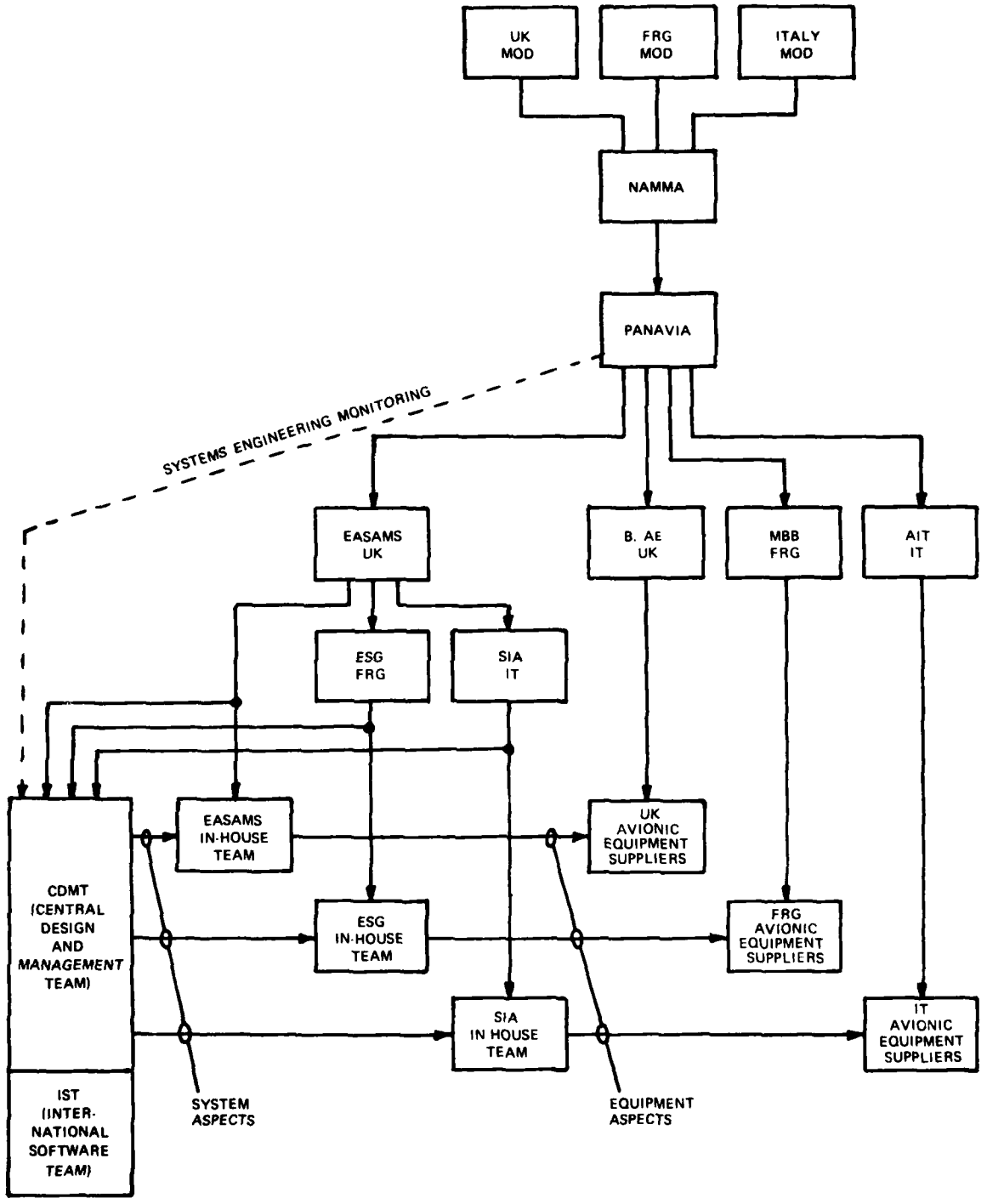


FIG. 1 MANAGEMENT AND INDUSTRIAL ORGANISATION (AVIONICS)

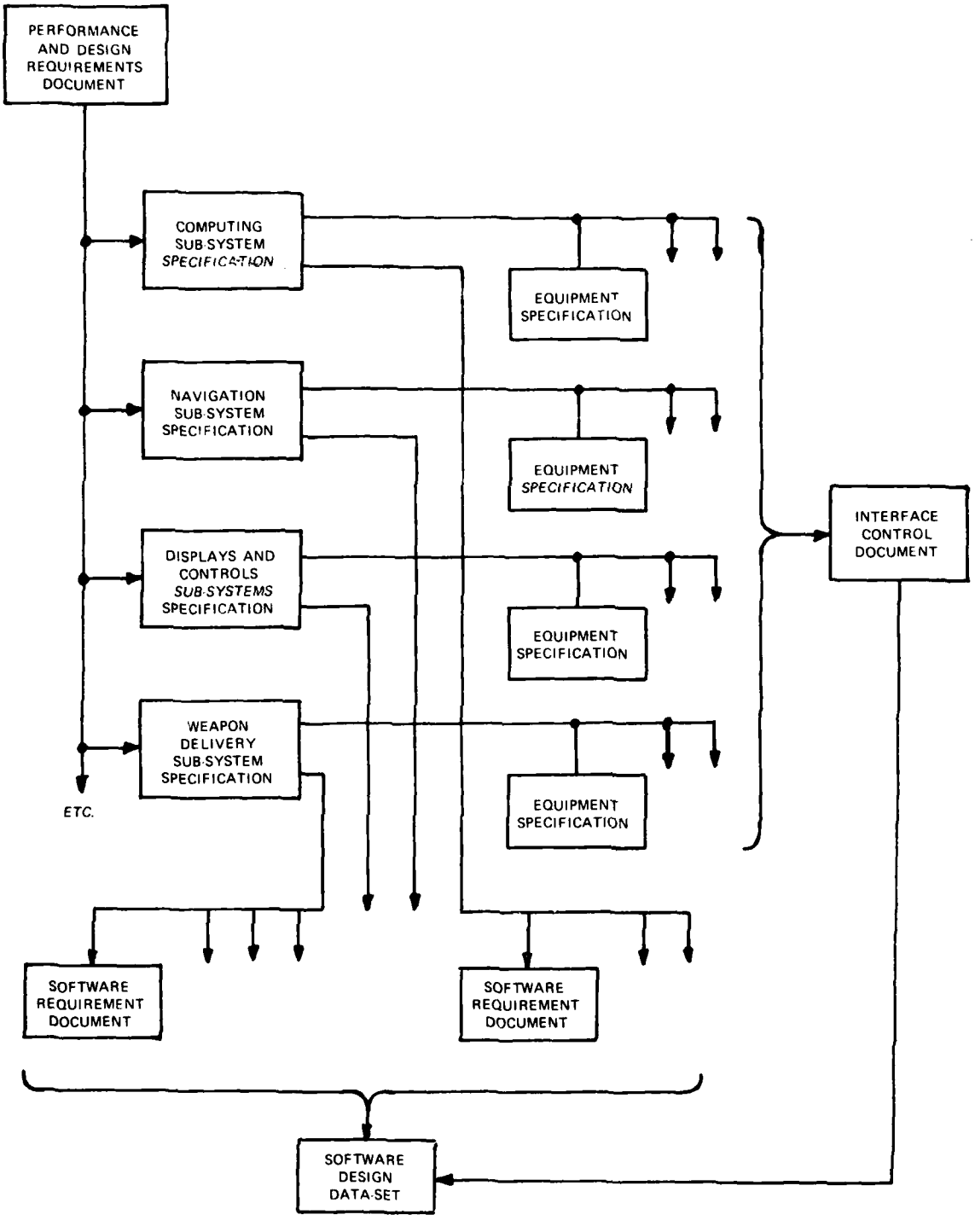


FIG 2. AVIONIC DESIGN REQUIREMENT SPECIFICATION

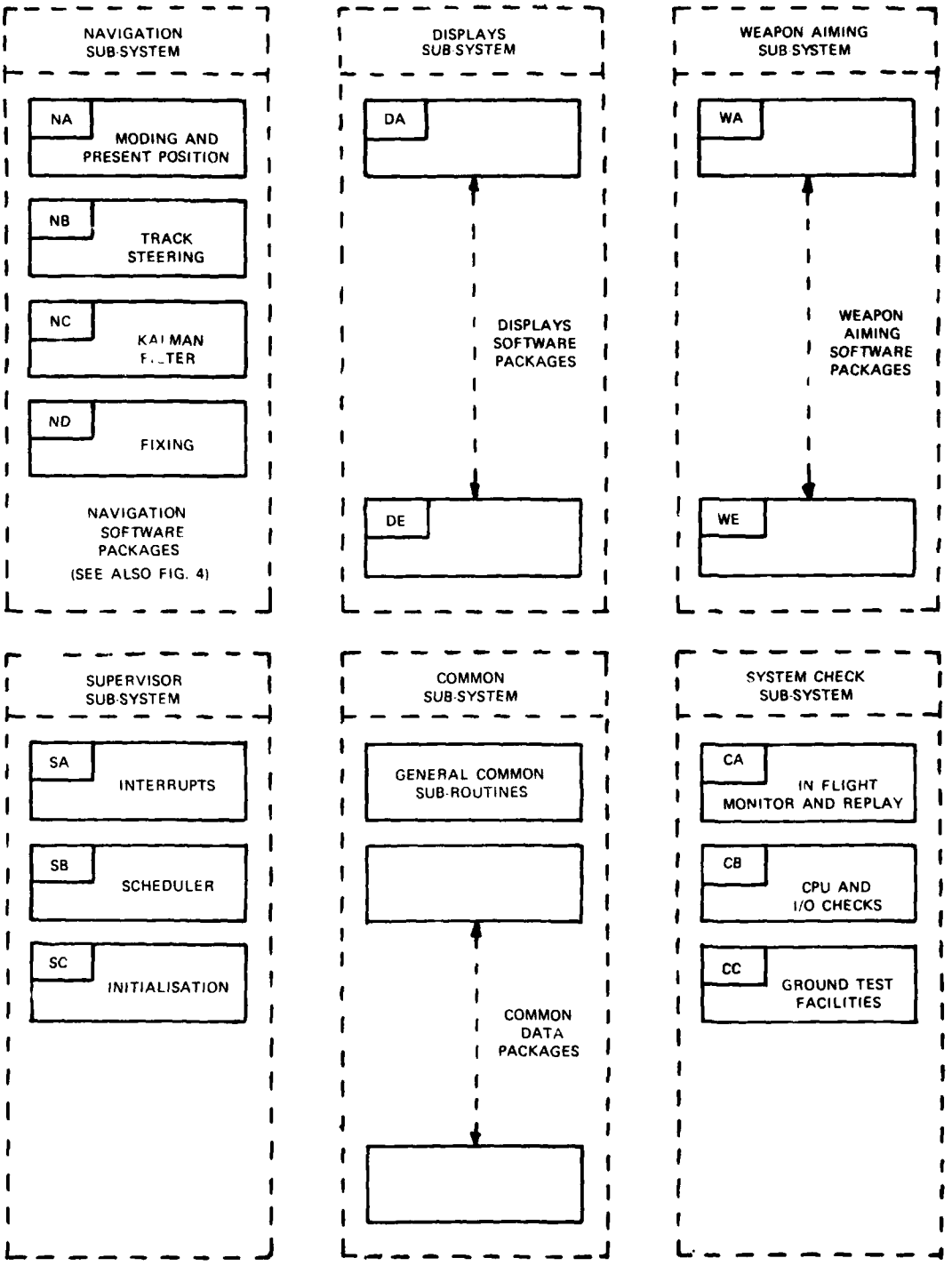


FIG. 3 OPERATIONAL FLIGHT PROGRAM STRUCTURE

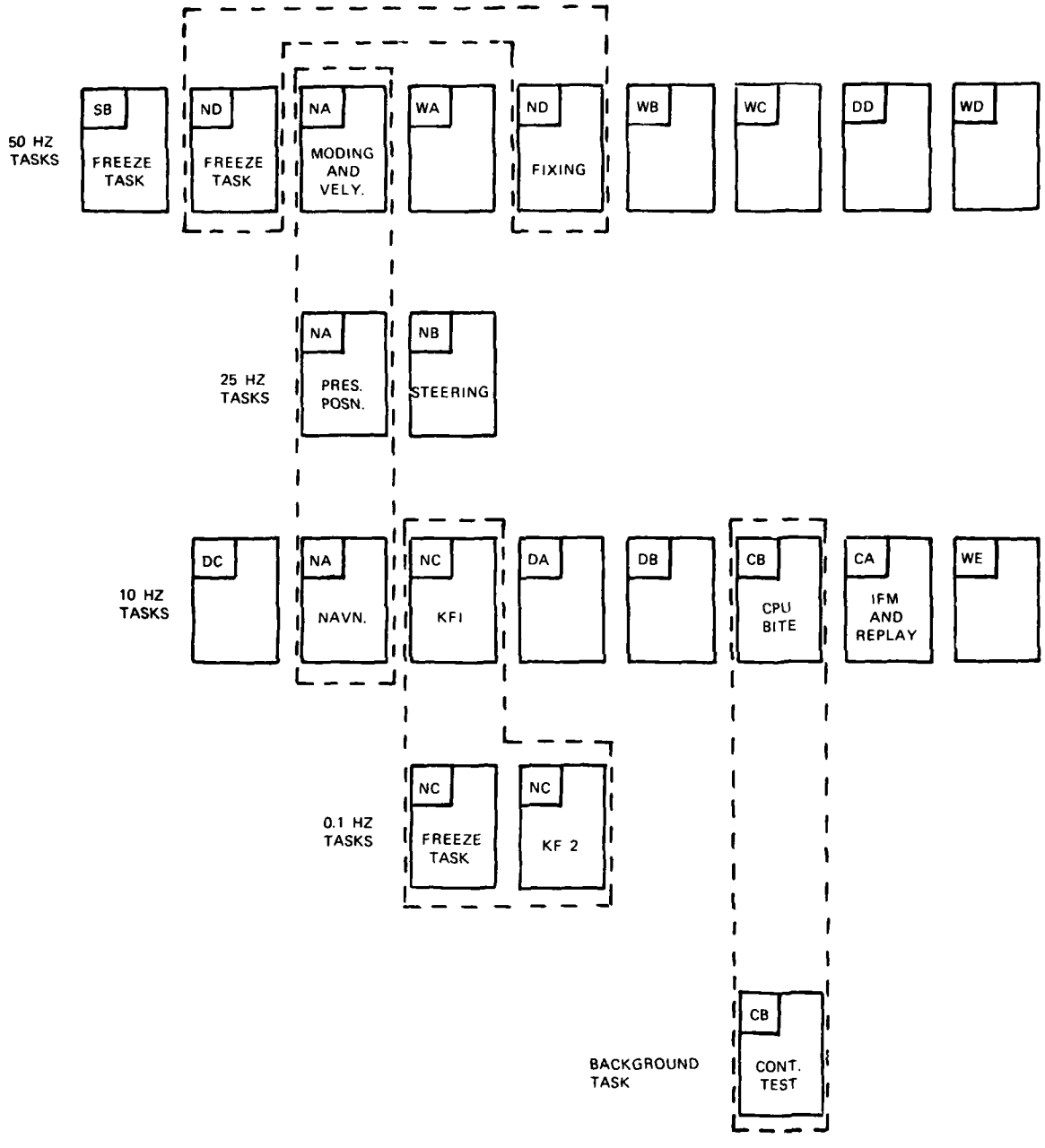


FIG. 4 PACKAGE SUB-DIVISION INTO TASKS OF DIFFERING ITERATION RATE.

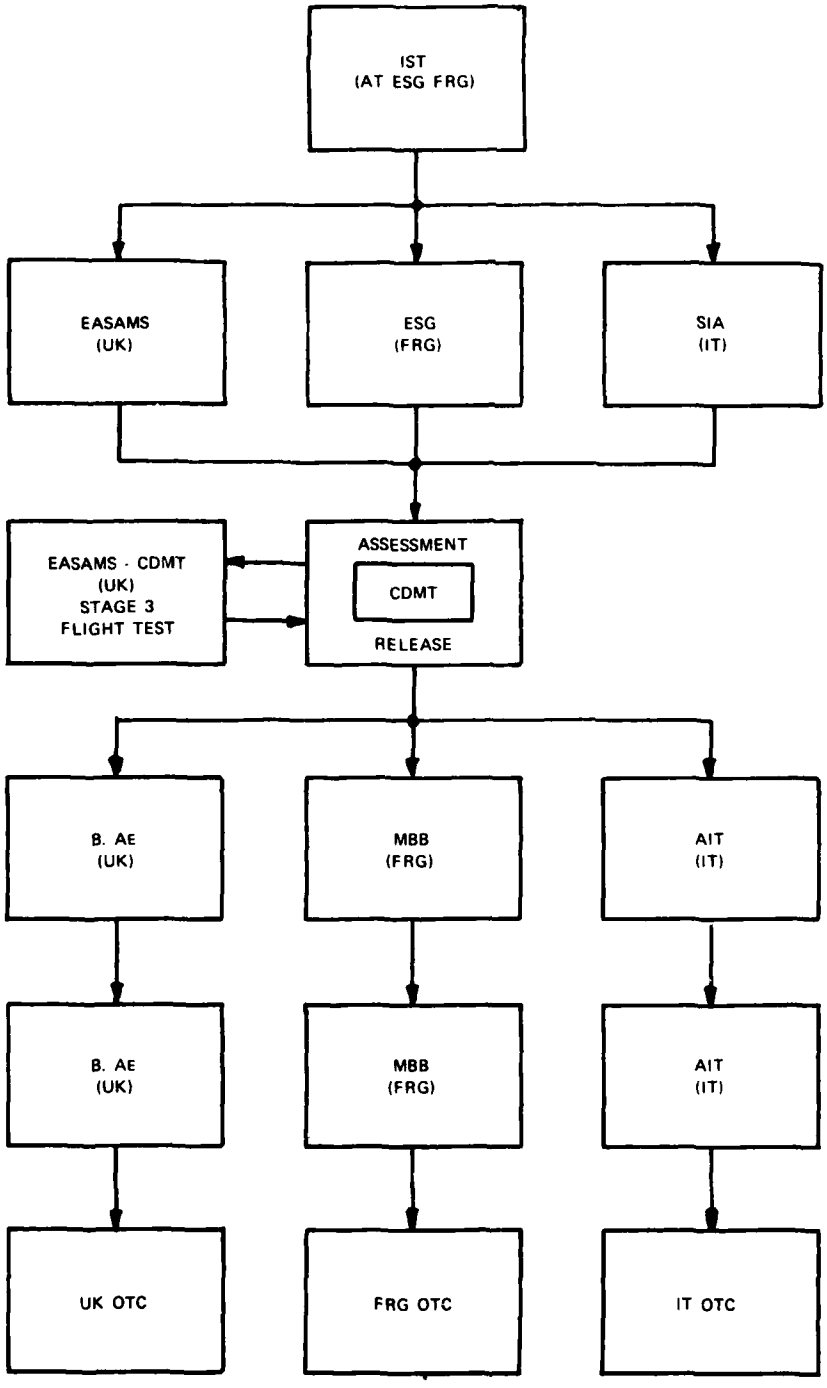


FIG. 5 SOFTWARE TEST AND INTEGRATION FACILITIES.

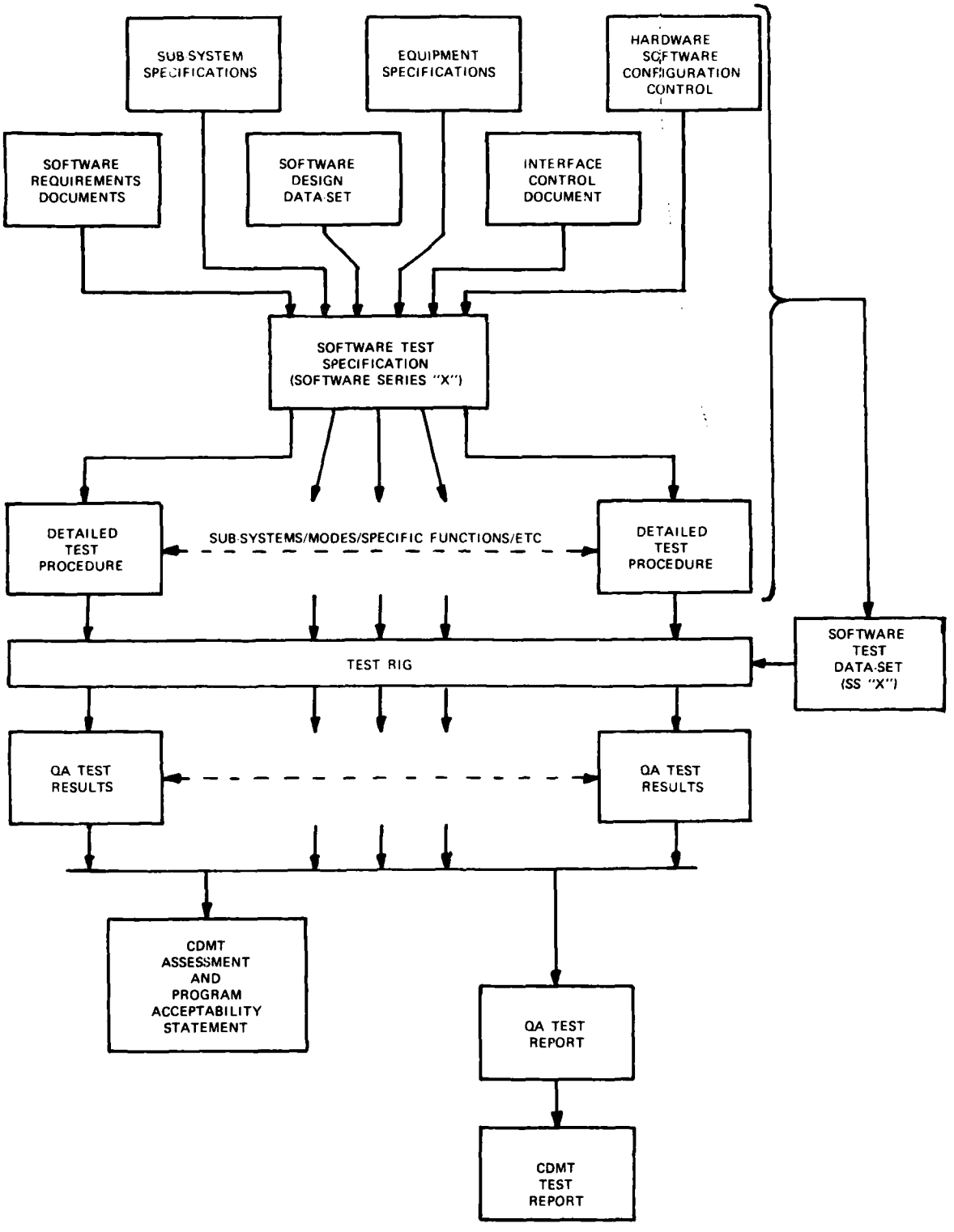


FIG. 6 SOFTWARE TEST DOCUMENTATION

LOGICIEL DU SYSTEME DE COMMANDE DE VOL ELECTRIQUE EXPERIMENTE SUR CONCORDE

Y. NEGRE : Direction des Etudes Avions
J. RAULLET : Direction des Etudes Avions
SOCIETE NATIONALE INDUSTRIELLE AEROSPATIALE
TOULOUSE 31.053 FRANCE

RESUME

Cet exposé rappelle brièvement le cadre de l'expérimentation réalisée sur l'avion supersonique CONCORDE d'un système de commande de vol électrique numérique destinée au contrôle de l'instabilité longitudinale. Après avoir souligné les objectifs de sécurité et de performances qui ont présidé à la définition du système, il est traité spécifiquement des problèmes relatifs à la définition, mise au point et expérimentation du logiciel utilisé. Des conclusions sont tirées principalement au niveau de la sécurité que l'on peut attendre de logiciels embarqués.

1 - CADRE DE L'EXPERIMENTATION EN VOL

C'est en 1974 qu'il fut décidé que l'AEROSPATIALE entreprendrait, sous contrat du Gouvernement Français l'étude, la réalisation et l'expérimentation en vol d'un système de commande de vol électrique numérique.

L'objectif retenu fut essentiellement de mettre au point des lois de pilotage permettant de voler à des centrages en arrière du foyer aérodynamique dans l'intérêt évident d'améliorer la finesse à basse vitesse comme indiqué Figure 0. Par ailleurs, il fut décidé que la conception du système à expérimenter devrait être, dans son architecture et sa technologie, suffisamment élaborée pour être généralisable à d'autres domaines intéressant le contrôle actif tels que le contrôle des charges aérodynamiques sur la voilure, le contrôle de cambrure de l'aile et l'augmentation des vitesses limites de flottement.

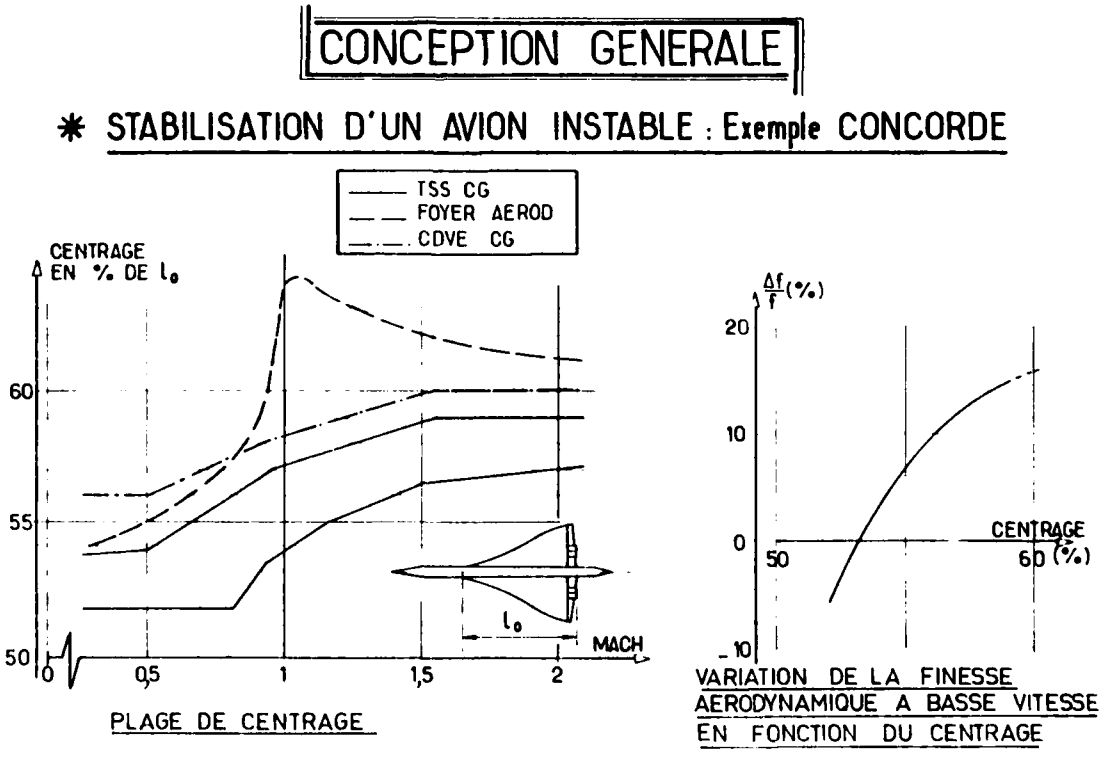


Figure 0

L'avion d'expérimentation retenu fut l'avion supersonique CONCORDE parce qu'étant un avion à grand domaine de vol et aussi parce que déjà équipé d'éléments propres à des commandes de vol électriques (servo-commandes électrohydrauliques par exemple). Il est d'ailleurs intéressant de noter au passage que les avions de série CONCORDE possèdent déjà un système de commande de vol électrique comme mode normal de pilotage de tous les jours dont le schéma est donné figure 1. On remarque la présence de deux chaînes de commande qui permettent de transmettre électriquement la position des organes de pilotage aux servo-commandes. Une commande mécanique est prévue comme secours des chaines électriques. En dernier secours, pour couvrir le cas

de blocage au niveau du pied de manche, il existe une dernière chaine électrique fonctionnant par détection des efforts au manche. L'examen de ce schéma va permettre de mieux faire ressortir les caractéristiques des nouvelles commandes de vol électriques expérimentées, dont nous donnons une description ci-après.

Le principe est donné figure 2. On constate que :

- les lois d'effort au manche sont obtenues directement par asservissement de la réponse avion aux efforts sur les commandes
- les systèmes de sensations artificielles, compensateurs d'efforts et stabilisateurs sont intégrés aux commandes électriques dans une technologie numérique
- la chaine de secours mécanique a disparu et l'organe de commande est miniaturisé pour tirer le plein bénéfice de la suppression de la commande mécanique
- on dispose de quatre chaines électriques, dont trois identiques, la quatrième devant être dissemblable pour des raisons de sécurité.

CONCEPTION GENERALE

PRINCIPE DES C.D.V.E. DE CONCORDE ACTUEL

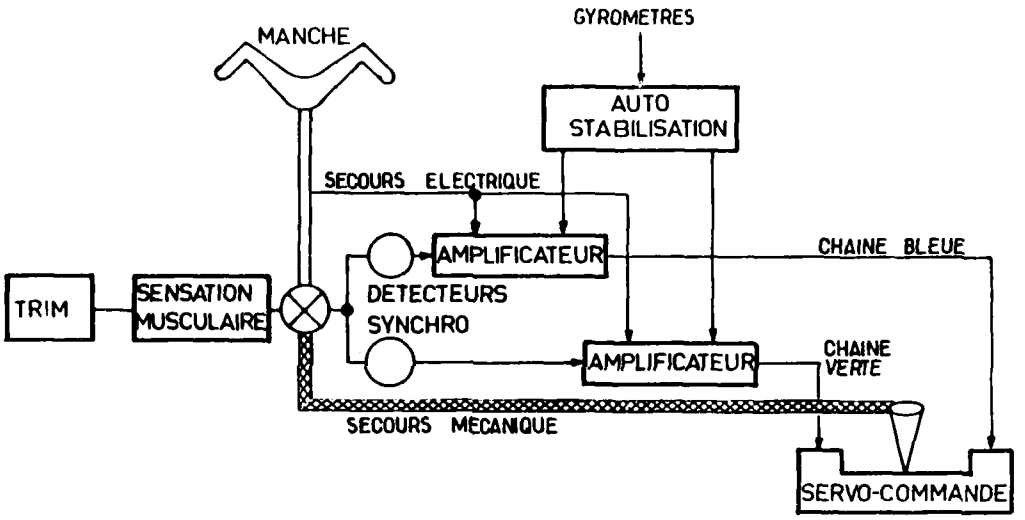


Figure 1

CONCEPTION GENERALE

PRINCIPE DE C.D.V.E GENERALISABLE

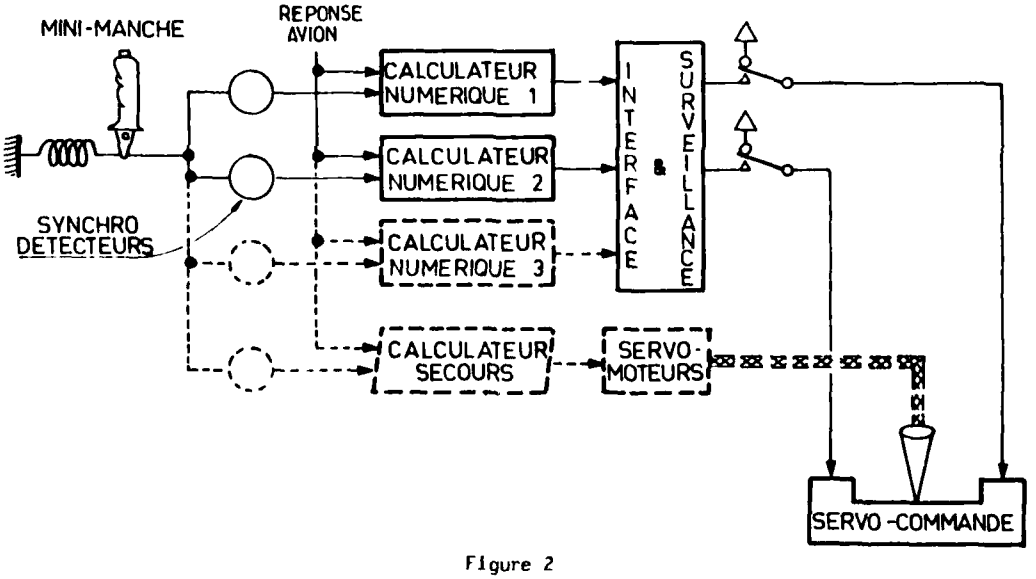


Figure 2

2 -NIVEAU DE SECURITE REQUIS

Il dépend des conséquences des évènements que l'on considère. Les règlements de certification établissent en général une classification des évènements en fonction de leurs conséquences. Pour chacune des conséquences il est accepté une probabilité d'apparition de l'évènement.

Le système de commande de vol électrique et plus généralement les systèmes intéressant le contrôle actif peuvent présenter des modes de pannes ayant des conséquences catastrophiques, de ce fait il est considéré que la probabilité de telles pannes doit être extrêmement improbable, c'est-à-dire, inférieure ou égale à 10^{-9} par heure de vol.

Parmi ces pannes on doit essentiellement considérer :

- la perte totale de commande
- l'embarquement non détecté des gouvernes.

Chacun de ces évènements conditionnant fortement l'architecture et la sécurité du système.

2.1 -Perte totale de commande

C'est à partir de la considération de cet évènement que l'on détermine le nombre minimal de chaînes électriques.

En considérant un M.T.B.F. de 1000 heures pour chaque chaîne (en incluant les détecteurs), l'objectif de 10^{-9}/heure conduit nécessairement à quatre chaînes indépendantes. La quatrième chaîne est technologiquement dissemblable afin de mieux se préserver de phénomènes susceptibles d'affecter plusieurs chaînes tels que foudroiements ou interférences électriques. A noter que cette redondance permet le décollage avec une chaîne en panne.

2.2 -Embarquement non détecté des gouvernes

C'est à partir de la considération de cet évènement que l'on détermine le niveau d'intégrité de chaque chaîne ; en effet, la redondance des chaînes ne suffit pas à se préserver d'un tel évènement. Il est nécessaire d'avoir la garantie que la ou les chaînes en fonctionnement auront la capacité de détecter leur propre panne avant propagation sensible du défaut.

Pour atteindre le niveau de sécurité fixé, il est nécessaire de faire appel à une panoplie de protections utilisant des techniques d'auto-surveillance en ligne de chaque chaîne, associées à des surveillances inter-chaînes, des surveillances externes et des arrangements appropriés entre les calculateurs, les servo-commandes et les gouvernes.

L'importance de ces protections mérite que l'on développe quelque peu leur mérite respectif.

2.3 -Types de surveillance

2.3.1 -Auto-surveillance en ligne

Ce type de surveillance est très largement utilisé sur des systèmes analogiques. Chaque chaîne assure sa propre protection en vérifiant le bon fonctionnement de ces éléments : capteurs, calculateurs et actuateurs. Diverses techniques de surveillance sont utilisées selon la nature de l'élément ; essentiellement on distingue :

- des comparaisons entre signaux de sortie et signaux d'entrée dans le cas de boucles d'asservissement
- des duplications et comparaisons des circuits avec des points consolidés.

Quelle que soit la technique utilisée, on peut formuler, dans le cadre du calcul analogique, les 3 remarques ci-après :

- l'introduction de surveillance se traduit toujours par l'adjonction physique de circuits propres à cette surveillance. A la limite, la complexité d'un calculateur peut être multipliée par un facteur supérieur à deux.
- On a pu, par analyse théorique, démontrer des niveaux de probabilité de pannes non détectées 10^{-9}/heure.
- L'expérience réelle, à ce jour, nous incite à une certaine prudence vis à vis des niveaux de sécurité démontrés.

Que deviennent ces remarques lorsqu'on introduit le calcul numérique ?

De par la nature du calcul numérique qui utilise une unité centrale de calcul capable d'exécuter toutes les opérations nécessaires à la réalisation de chaque fonction, on peut imaginer de réaliser les fonctions de surveillance par programmation sans adjonction de circuits particuliers. Cette voie fut explorée par l'AEROSPATIALE à l'occasion de l'expérimentation du système de commande de vol électrique. A cette occasion, on développa au niveau logiciel des instructions de test dans le but de vérifier à tout instant le fonctionnement du calculateur et en particulier le fonctionnement de l'unité centrale de calcul.

La conclusion fut qu'on ne pouvait pas espérer démontrer des niveaux de probabilité de pannes non détectées à mieux que 10^{-6}/heure de fonctionnement en utilisant seulement une surveillance par programmation. Dès lors, pour atteindre le niveau requis de 10^{-9}/heure il devient nécessaire de faire appel, comme en analogique, à des duplications de circuits et en particulier la duplication de l'unité centrale de calcul.

En résumé, l'utilisation du numérique permet, de par sa nature, d'assurer une auto-surveillance sans augmentation importante de la complexité jusqu'au niveau de 10^{-6}/heure. Au delà, il est nécessaire d'accroître la complexité et les remarques faites pour les systèmes analogiques s'appliquent aux systèmes numériques avec, nous le verrons plus loin, un problème spécifique au calcul numérique qui est la sécurité relative au logiciel.

2.3.2 -Surveillance inter-chaînes

Cette surveillance consiste à comparer deux ou plusieurs chaînes. Une comparaison s'exerçant entre deux chaînes permet de détecter la panne d'une chaîne sans possibilité d'identification de la chaîne en panne.

Pour assurer la survie du système, il est nécessaire de disposer d'au moins trois chaînes.

Cette surveillance inter-chaînes peut se faire, soit par des comparateurs logiques, soit par des dispositifs à voteurs.

2.3.3 -Surveillance externe

Il s'agit de protections globales externes au système et basées généralement sur la détection de mouvementsexcessifs avion tels que le facteur de charge, la vitesse angulaire des roulis, etc.. Le mérite de ce genre de protection, au niveau sécurité, est remarquable car on exerce une surveillance hors tout en bout de chaîne sur le mouvement avion, ce qui assure une indépendance intrinsèque entre une panne possible et sa détection.

Hélas, à moins de régler les seuils de détection à des valeurs qui deviennent excessives pour la sécurité, ce genre de protection conduit à des déclenchements intempestifs, hors pannes, sur des manoeuvres rapides ou en vol très turbulent.

2.3.4 -Arrangement entre calculateurs, servo-commandes et gouvernes

On peut, par des arrangements appropriés, obtenir au niveau des gouvernes, une sécurité supplémentaire par sommations en force des ordres délivrés par plusieurs chaînes.

2.4 -Architecture retenue pour l'expérimentation en vol

L'objectif de l'expérimentation n'étant pas de certifier le système de commande de vol électrique, le nombre de chaînes électriques fut volontairement limité à deux avec comparaison, la survie étant assurée, après panne, par les chaînes normales de commande de vol, côté pilote, de CONCORDE. Le schéma du système et de ses surveillances est donné figure 3.

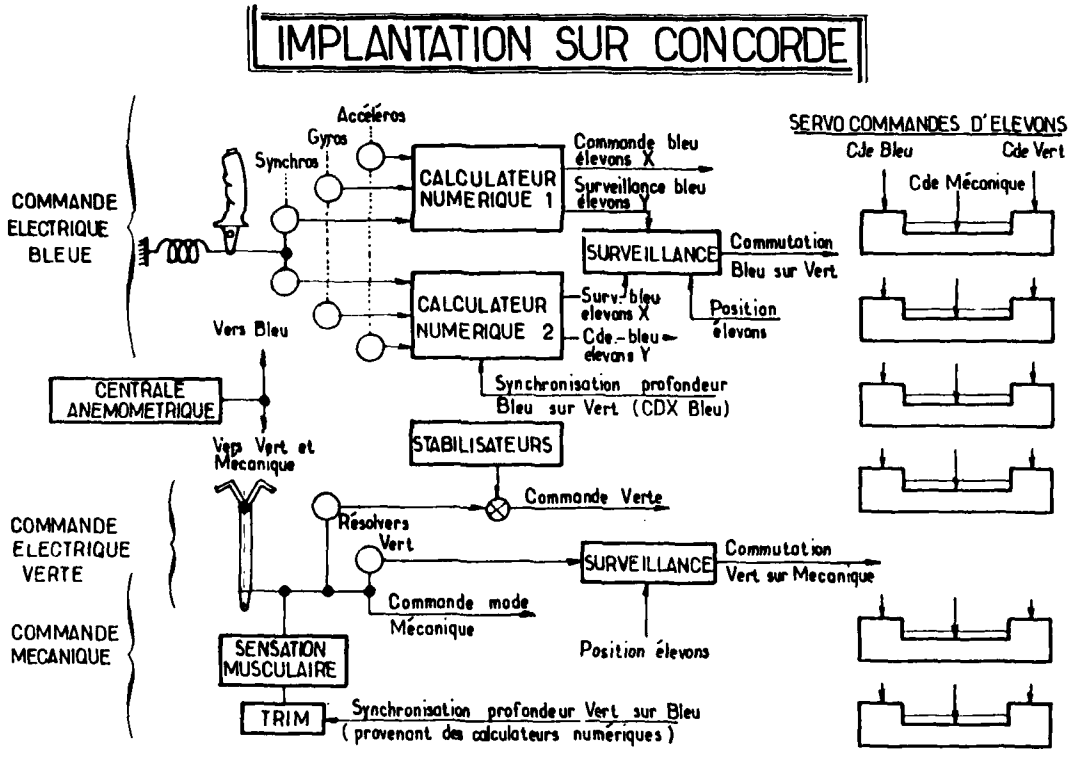


Figure 3

3 -DEFINITION ET EXPERIMENTATION DU LOGICIEL UTILISE

3.1 -Matériel

Avant de décrire le logiciel, il est nécessaire de donner une description succincte du calculateur sur l'aspect matériel.

Le calculateur retenu fut un calculateur du type spécifique fabriqué par THOMSON-CSF par opposition aux calculateurs du type universel qui, à l'époque, furent largement développés par ailleurs.

Il se caractérise au niveau matériel par :

- la présence de deux unités centrales de calcul, travaillant en parallèle, l'une consacrée à des opé-

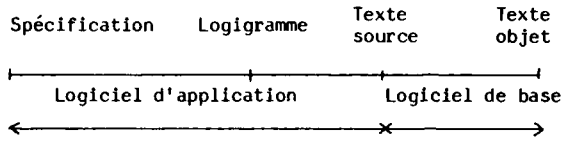
rations courtes du type addition, soustraction, etc.., l'autre consacrée à des opérations longues du type : racine carrée, rotation de vecteur, multiplication, division, etc...

- un mode de transmission de données série sur 16 bits
- un accès direct mémoire pour les échanges avec l'extérieur permettant une gestion autonome des entrées, sorties indépendamment des unités de calcul
- un certain nombre de mémoires :
  - . 10 K octet de mémoire programme
  - . 128 mémoires RAM de travail de 16 bits
  - . 128 mémoires RAM de travail de 1 bit
  - . 128 mémoires tampons de sortie de 16 bits
  - . 128 mémoires tampons de sortie de 1 bit
  - . 128 mémoires tampons d'entrée de 1 bit
  - . 128 mémoires tampons d'entrée de 16 bits
- des échanges directs entre le calculateur et les entrées/sorties ; le calculateur se contentant d'écrire ou de lire dans des mémoires tampons. Les informations de ces mémoires sont exploitées ou rafraichies sans blocage du calculateur.

3.2 -Logiciel

La règle essentielle qui présida à la réalisation du logiciel fut celle de la simplicité. On chercha en particulier à ce que les différentes étapes de la programmation soient contrôlables et directement exploitables par des Techniciens systèmes non spécialistes en programmation numérique.

Ces différentes étapes apparaissent dans le diagramme ci-dessous :



3.2.1 -Logiciel de base

Le logiciel de base se caractérise par :

. La génération de la bande objet, obtenue à partir d'une bande source et au travers d'un assembleur réalisé sur un calculateur extérieur HONEYWELL DDP 124 en utilisant le logiciel de base de ce calculateur.

Sur le listing généré, ou bande objet, on trouve :

- le texte source avec numérotage des lignes, ce qui permet une correction aisée des bandes sources
- le texte objet avec l'adresse mémoire qui donne exactement ce que l'on devra trouver en mémoire
- les erreurs de synthaxe

Par ailleurs, l'assembleur :

- permet l'affectation automatique des adresses des données permanentes
- permet de rentrer des valeurs numériques en octal, en entier, en décimal cadré
- possède des pseudo-instructions et des pseudo-opérations
- permet l'utilisation de commentaires.

. Un programme de correction de bandes

. Un programme d'aide à la mise au point,obtenu avec l'aide d'un calculateur extérieur INTELLEC 8, couplé à un télétype et une perforatrice rapide capable de lire les mémoires du calculateur, celui-ci étant fermé.

Le point marquant de ce logiciel de base est le nombre réduit d'instructions apparaissant sur le texte objet écrit en langage machine. Cette caractéristique provient :

- de la présence, déjà signalée, d'instructions puissantes et bien adaptées, câblées dans les deux unités centrales de calcul
- de l'absence d'interruption grâce aux échanges directs entre le calculateur et l'extérieur.

Il en résulte qu'il y a peu de différence entre le texte objet et le texte source écrit en langage assemblage et il est très aisé de contrôler le passage de l'un à l'autre.

3.2.2 -Logiciel d'application

C'est le passage de la spécification du calculateur au programme (texte source) qui doit être utilisé pour le logiciel de base. On distingue trois étapes :

- la rédaction des spécifications des systèmes qui donne lieu à une représentation analogique de fonction à réaliser
- la réalisation du logigramme qui est la description logique des fonctions à réaliser
- l'écriture de ce logigramme sous forme de programme.

La figure 4, à partir d'un exemple intéressant l'axe latéral, montre comment à partir d'une représentation analogique de circuits découlant directement de la spécification, on obtient un programme écrit en langage assemblage. La symbologie représentant les instructions disponibles est

donnée figure 5.

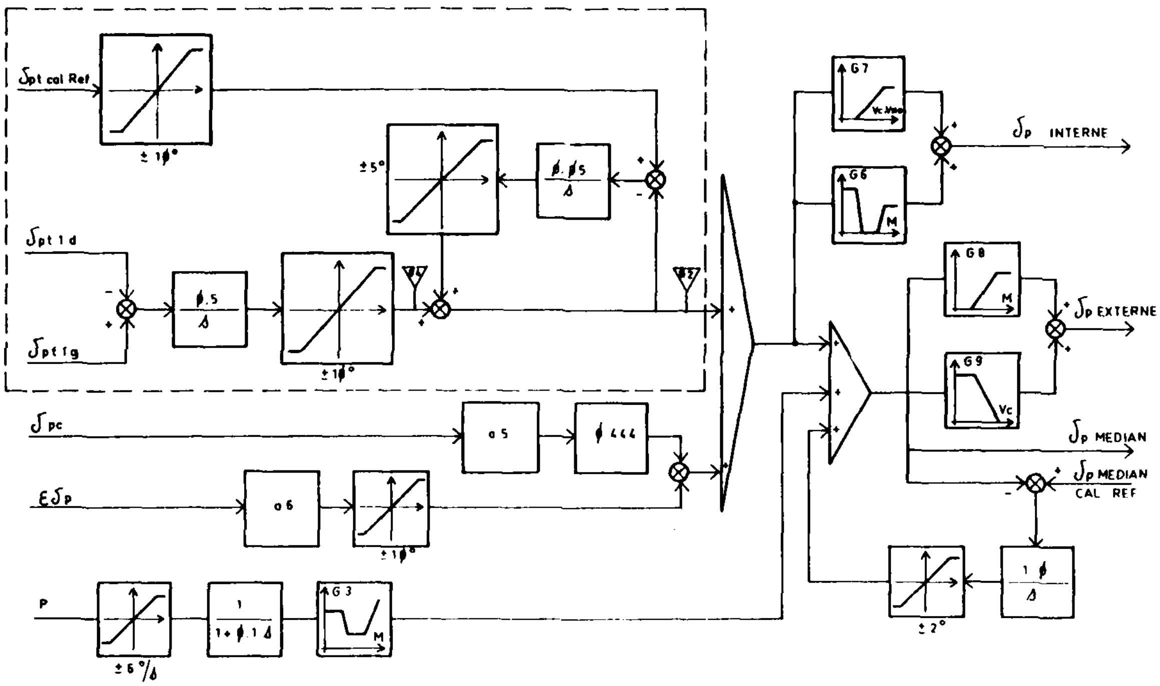


Figure 4

REPERTOIRE D'INSTRUCTION

INSTRUCTIONS	SYMBOLOGIE	REALISATION
ADD	A, B → [+ ≶n] → C	C = (A+B)2^{±n}
SOU	A, B → [− ≶n] → C	C = (A−B)2^{±n}
ABS	A, B → [|+≶n] → C	C = (A+|B|.signe de A)2^{±n}
DA 1	A, B → [DA 1] → b, c	C = A+B b = 1 si débordement ΔY = b.signe de C
DA 2	A, B → [DA 2] → b, c	C = A+B+ΔY b = 1 si débordement ΔX = b.signe de C
DA 3	A, B → [DA 3] → b, c	C = A+B+ΔX b = 1 si débordement
ALT	A, B, b → c	C = A si b = 1 C = B si b = 0
SAT	A, B → b, c	C = A et b = 0 si |A| ≤ |B| C = B et b = 1 si |A| > |B|
LCP	A, B → [LCP] → b	b = 1 si A > B
LET	a, b → c	c = a.b
LOU	a, b → c	c = a+b

INSTRUCTIONS	SYMBOLOGIE	REALISATION
LOX	a, b → c	c = a ⊕ b
JET JOU JOX JCP	idem LET, LOU, LOX avec la lettre J à l'intérieur du symbole a, b, adresse A, B, Adresse	Saut à l'adresse indiquée si a.b = 1 / a+b = 1 / a ⊕ b = 1 / A > B
MUL	A, B → [x ≶n] → C	C = (A.B) 2^{±n}
DIV	A, B → [: ≶n] → C	C = (A:B) 2^{±n}
RAC	A → [√ ≶n] → R	R = √A 2^{±n}
ROT	Xo, Yo, θ → R → X, Y	X = Xo cos θ + Yo sin θ Y = Yo cos θ − Xo sin θ
VEC	Xo, Yo → θo, V, ψ	V = $\sqrt{X_0^2 + Y_0^2}$ ψ = θ_0 + Arc tg $\frac{Y_0}{X_0}$
FPR		FIN DE PROGRAMME

Figure 5

Le logigramme et le listing correspondants sont donnés figures 6 et 7.

Il est important de remarquer au niveau du logigramme, que la lecture du logigramme permet à la fois de :

-retrouver la représentation analogique découlant des spécifications. Ceci parce que le logigramme respecte la chronologie des évènements et donne, en fait, le déroulement des opérations dans le temps sans gestion d'interruptions grâce aux échanges directs avec l'extérieur.

-d'écrire directement le programme. Ceci parce que les instructions utilisées sont particulièrement bien adaptées à la résolution des fonctions à réaliser et parce que l'on a décomposé les opérations à réaliser en modules de calcul, chaque module représentant une opération longue et trois opérations courtes.

Cette "transparence" amont et aval du logigramme est une des caractéristiques essentielles de ce logiciel d'application.

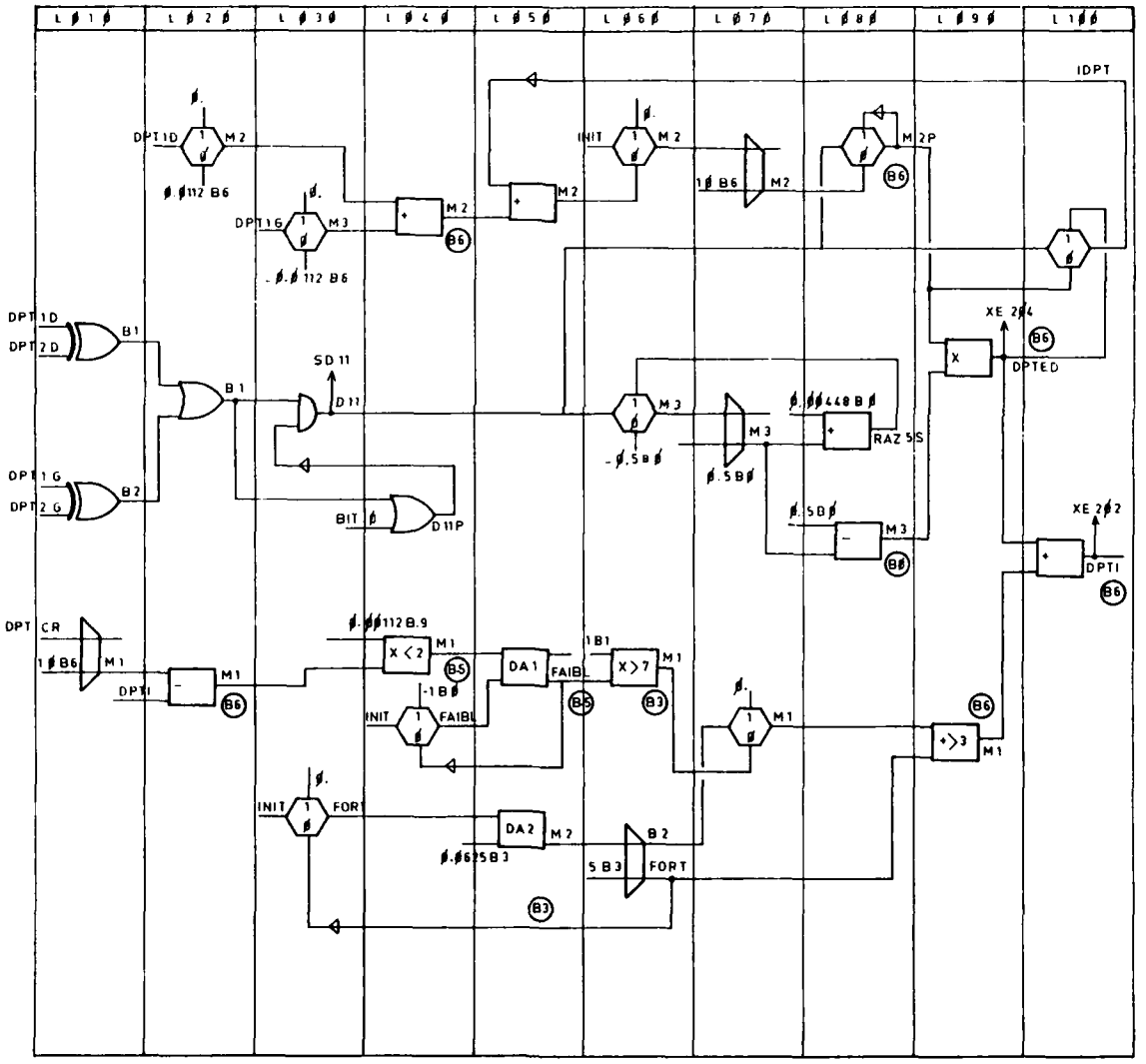


Figure 6

```
                                *          COMMANDES DE VOL NUMERIQUES                                PAGE 1


0001                            *          COMMANDES DE VOL NUMERIQUES
0002                            *
0003                            *
0004                            *          PLANCHE L
0005                            *
0006                            *          ACQUISITION ET SURVEILLANCE DES ENTREES DISCRETES TRIM
0007                            *
0008                            *          CALCUL DE L'ORDRE DE TRIM
0009                            *
0010                            *
0011                            *
0012        00                  DPT1D EGL 00 EB1                    ENTREE DISCRETE TRIM DROIT
0013        24                  DPT1G EGL 11 EB1                      "       "     "  GAUCHE
0014        E8                  DPT2D EGL 72 EB1                      "       "     "  DROIT SURVEILLANCE
0015        C0                  DPT2G EGL 60 EB1                      "       "     "  DROIT SURVEILLANCE
0016        14                  INIT  EGL 05 EB1                    MONTEE D'ALIMENTATION
0017        09                  BIT0  EGL 02 TB1                    BIT A ZERO
0018        34                  DPTCR EGL 15 EM1                    BRAQUAGE DE TRIM DU CALCULATEUR DE REFERE
0019        30                  SD11  EGL 17 SB1                    SORTIE DISCRETE D11.
0020        66                  XE202 EGL 31 SM2                    SORTIE ANALOGIQUE POUR ENREGISTREMENT
0021        6E                  XE204 EGL 33 SM2                    SORTIE ANALOGIQUE POUR ENREGISTREMENT
0022                            *
0023                            *
0024                            *
0025        EF F7 F3 FB FF EB           DPM IDPT,M2P,RAZSS,FAIBL,FORT,DPT1
0026        FF                          DPB D11P
0027                            *
0028                                    FPG


                                *          COMMANDES DE VOL NUMERIQUES                                PAGE 2

0029                            *
0030 00000  07 01 01 01         L010  NOP
0031 00004  F7 00 EB 05                 LOX DPT1D,DPT2D,B1
0032 00010  F7 24 C0 09                 LOX DPT1G,DPT2G,B2
0033 00014  E4 14 0014 01 05            SAT DPTCR,10B6,POUB,M1
     00022  02 01
0034 00024  07 01 01 01         L020  NOP
0035 00030  81 0000 0000 00 09          ALT 0,0.0112B6,DPT1D,M2
0036 00037  87 05 09 05                 LOU B1,B2,B1
0037 00043  86 05 E7 02 05              SOU M1,DPTI,M1
     00050  02 01
0038 00052  07 01 01 01         L030  NOP
0039 00056  81 0000 F6FF 24 0D          ALT 0,-0.0112B6,DPT1G,M3
0040 00065  7F 05 FB 0D 3D              LET B1,D11P,D11,#SD11
0041 00072  82 0000 FB 14 FF            ALT 0,FORT,INIT,FORT
     00100  02 01
0042 00102  03 3849 01 05       L040  MUL 0.00112B-9,M1,C2,M1
0043 00107  47 09 0D 02 09              ADD M2,M3,M2
0044 00114  87 05 09 FF                 LOU B1,BIT0,D11P
0045 00120  82 0080 F7 14 FB            ALT -1B0,FAIBL,INIT,FAIBL
     00126  09 05
0046 00130  07 01 01 01         L050  NOP
0047 00134  47 EB 09 02 09              ADD IDPT,M2,M2
0048 00141  27 05 FB 01 FB              DA1 M1,FAIBL,POUB,FAIBL
0049 00148  44 FF 0001 01 09            DA2 FORT,0.0625B3,POUB,M2
     00154  02 01
0050 00156  03 0040 01 FB       L060  MUL 1B1,FAIBL,.7,M1
0051 00163  83 0000 09 14 09            ALT 0,M2,INIT,M2
0052 00171  7F EF 0000 0D 0D            ALT RAZSS,-0.5B0,D11,M3
0053 00177  E4 09 0050 09 FF            SAT M2,5B3,B2,FORT
     00205  1E 05
0054 00207  07 01 01 01         L070  NOP
0055 00213  E5 09 0014 01 09            SAT M2,10B6,POUB,M2
0056 00221  E5 0D 0040 01 0D            SAT M3,0.5B0,POUB,M3
0057 00227  82 0000 05 09 05            ALT 0,M1,B2,M1
     00235  02 01
0058 00237  07 01 01 01         L080  NOP


                                *          COMMANDES DE VOL NUMERIQUES                                PAGE 3

0059 00243  87 F3 09 0D F7              ALT M2P,M2,D11,M2P
0060 00250  43 4200 0D 02 F3            ADD 0.0044B0,M3,RAZSS
0061 00258  C2 0040 0D 02 0D            SOU 0.5B0,M3,M3
     00264  02 01
0062 00266  04 F7 01 0D         L090  MUL M2P,M3,DPTED,#XE204
0063 00273  46 05 FF 0E 05              ADD M1,FORT,.1,M1
     00277  6E 11
0064 00302  07 01 01 01         L100  NOP
0065 00306  87 11 F7 0D EF              ALT DPTED,M2P,D11,IDPT
0066 00313  4E 11 05 02 66 EB           ADD DPTED,M1,DPTI,#XE202
     00321  02 01
0067                            *
0068                                    FIN
```

Figure 7

3.2.3 -Sécurité du logiciel

En plus des opérations de contrôle du logiciel effectuées à chaque étape de sa réalisation et, comme nous l'avons vu, facilitées par un langage qui, in fine, même au niveau de la bande objet ou en langage machine reste compréhensible par un ingénieur système, on doit ajouter des contrôles de fonctionnement au niveau du calculateur et au niveau du système couplé à un simulateur de vol. La figure 8 donne un résumé du processus d'établissement du logiciel et des contrôles exercés. La figure 9 résume pour chacune des étapes de la programmation le type d'erreurs susceptibles d'être rencontrées et donne une estimation des probabilités de détection. On notera que pour certains types d'erreurs, on ne peut pas avoir la certitude qu'elles ne se produiront pas. C'est à partir de ces considérations qu'il fut décidé d'adopter une double programmation.

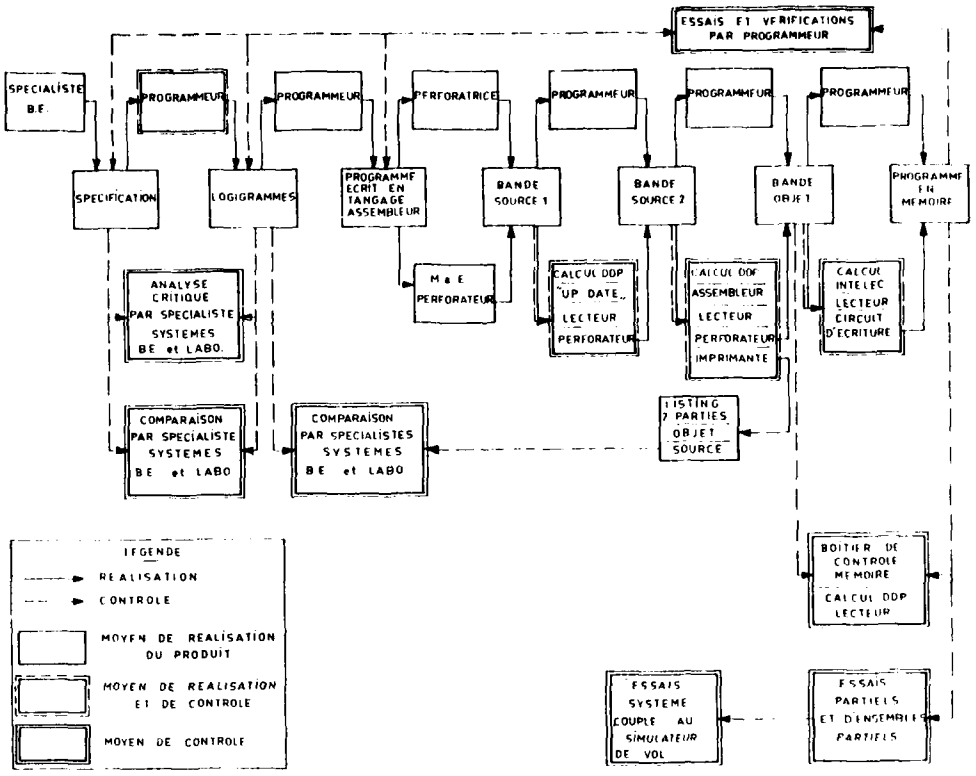


Figure 8

Coefficient de détection - 100% toutes les erreurs sont détectées - A -dº- sous réserve d'une défaillance humaine - B Toutes les erreurs sont probablement détectées - C Certaines erreurs, mais pas toutes, sont détectées	Analyse critique spécif. par Spécialistes BE/LABO	Analyse Spécif. par Programmeur	Génération de la source 2 par l'UP DATE	Opération d'assemblage	Opération de chargement Mémoire	Essais & vérifications par Programmeur	Essais partiels	Essais sur Simulateur	Contrôle chargement en Mémoire	Opération de désassemblage	Comparaison listing logigramme	Comparaison logigramme Spécific.
1/Ecriture de la Spécification												
a)Défaut d'analyse du Problème	B.B						B	B				
b)Défaut de présentation	B.B	B					B	B				
c)Insuffisance de définition	A.A	A					A	A				
2/Ecriture des logigrammes												
a)Erreur d'interprétation Spécification							A ou B	A ou B				A.A
b)Erreur de programmation							A ou B	A ou B				A.A
3/Ecriture en langage Assembleur												
a)Erreur de Syntaxe				100 %								
b)Autres erreurs				C		A	A ou B	A ou B			A.A	
4/Perforation bande "Source 1"												
a)Erreur de frappe				100 % ou C		A	A ou B	A ou B			A.A	
b)Erreur de perforation			0 à 100%	100 % ou C		A	A ou B	A ou B			A.A	
5/Perforation bande "Source 2"												
a)Erreur de lecture"Source 1"			0 à 100%	100 % ou C		A	A ou B	A ou B			A.A	
b)Anomalies de fonctionnement calculateur DDP				100 % ou C		A	A ou B	A ou B			A.A	
c)Erreur de perforation "source 2"				100 %		A	A ou B	A ou B			A.A	
6/Opération d'assemblage												
a)Erreur de lecture "source 2"				100 %		A	A ou B	A ou B		100 %	A.A	
b)Anomalies "Assembleur" et calculateur DDP						A	A ou B	A ou B		100 %	A.A	
c)Erreur de perforation "objet"					100 %	A	A ou B	A ou B		100 %		
d)Erreurs dues à l'imprimante (Partie source du listing)											A.A	
7/Chargement en Mémoire												
a)Erreur de lecture Objet"					100 %	A	A ou B	A ou B	100%	100 %		
b)Défaut calculateur INTELLEC et dispositif d'inscription					B	A	A ou B	A ou B	100%	100 %		
c)Mémoire défectueuse					100 %	A	A ou B	A ou B	100%	100 %		

Figure 9

Double programmation :

On a vu que l'architecture retenue pour l'expérimentation en vol comportait deux calculateurs avec une comparaison en sortie de ces calculateurs et ce afin de se protéger contre toutes pannes au niveau matériel, qui n'auraient pas été détectées par l'auto-surveillance propre à chaque calculateur. Auto-surveillance, on l'a vu, pour laquelle on avait pu atteindre un niveau de sécurité de 10^{-6}. Afin de se préserver contre une erreur éventuelle de logiciel qui aurait pu échapper à toutes les mesures de contrôle que nous venons de décrire, on effectua deux programmations avec deux équipes séparées, chacun des logiciels ainsi obtenu étant appliqué sur chaque calculateur. Ceci permit de détecter une erreur dans un des deux logiciels, erreur qui provenait d'une mauvaise interprétation de la spécification au niveau du logigramme.

4 -RESULTATS

Seulement 10 heures d'essais en vol furent nécessaires pour balayer la totalité du domaine de vol normal et explorer les cas de centrage arrière, principalement à basses vitesses, tel que montré sur la figure 10.

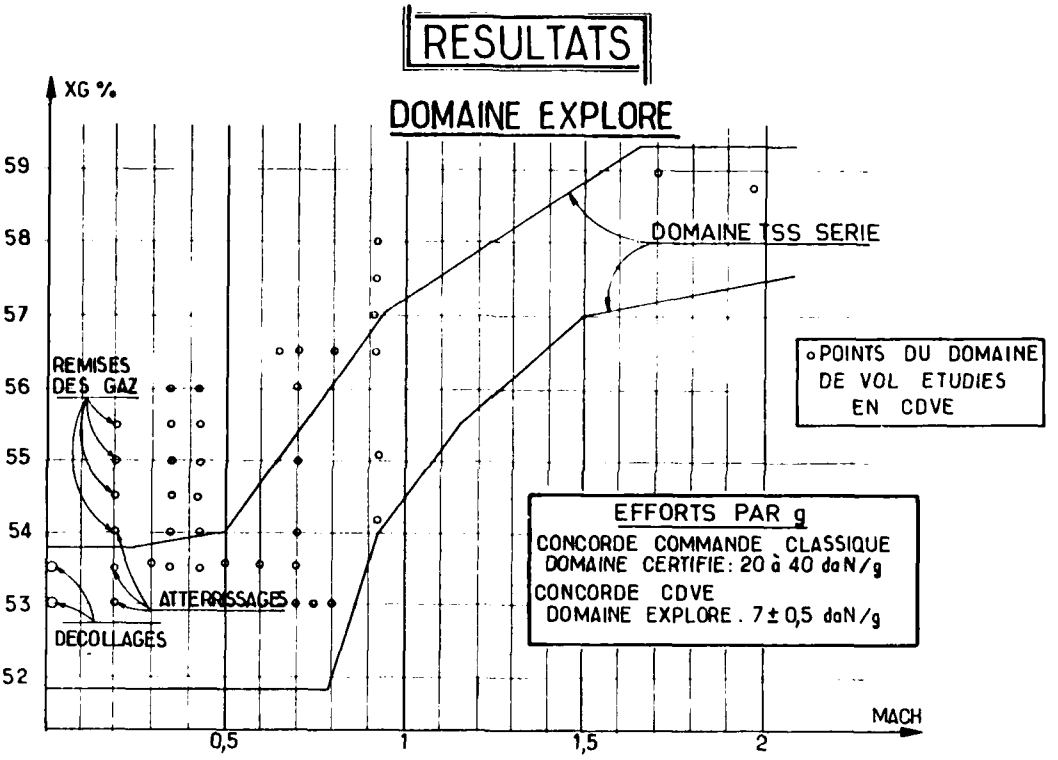


Figure 10

En ce qui concerne les résultats on retiendra :

4.1 -Au niveau avion

L'impression générale a été une très grande qualité de pilotage d'autant plus remarquable que la plage de centrage explorée était importante mais également que l'élément de comparaison était les commandes classiques de CONCORDE dont on connaît déjà la qualité intrinsèque.

Par ailleurs, piloter avec la main gauche sur un mini-manche n'était pas de nature à faciliter le pilotage.

Les efforts par g sont restés constants dans tout le domaine de vol à mieux de 10 %, alors qu'il est commun de les voir varier du simple au double sur n'importe quel autre avion.

De même, la dispersion du temps de réponse fût nettement améliorée. Quant à la stabilité en assiette, elle fut, par définition même du système, jugée comme quasi parfaite.

4.2 -Au niveau calculateur

Un excellent comportement du calculateur "COMVOL" caractérisé par l'absence de déconnexion intempestive (que l'on aurait pu craindre du fait de la comparaison de deux calculateurs) et une très bonne disponibilité (aucune panne en vol et une seule panne au sol du calculateur).

En outre, on peut résumer les avantages de ce calculateur en notant :

- la rapidité du temps de calcul : 22 ms
- la simplicité et facilité de programmation
- la transparence du logiciel

- le contrôle de chaque étape de la programmation aisé et accessible aux ingénieurs systèmes

Tous ces avantages découlent essentiellement des caractéristiques de conception au niveau matériel que nous rappelons ci-après :

- présence de deux unités centrales travaillant en parallèle
- rapidité des échanges avec l'extérieur ne nécessitant pas d'interruption dans le déroulement du programme
- "macro-instructions" câblées.

5 - CONCLUSIONS RELATIVES A LA SECURITE

Cette expérimentation a permis de maîtriser les problèmes posés par un système de commande de vol électrique capable d'assurer le vol avec des marges statiques largement négatives. Cependant, avant d'être en mesure de certifier un tel système sur un avion de transport civil, il nous paraît nécessaire de faire progresser le niveau de sécurité des systèmes dont les pannes peuvent avoir des conséquences immédiatement catastrophiques.

Ce problème de sécurité concerne aussi bien des systèmes numériques que des systèmes analogiques. Nous avons déjà noté que l'expérience réelle sur des systèmes analogiques conduisait à une certaine prudence sur les niveaux de sécurité démontrés. Il convient donc, dans un premier temps, de renforcer à la fois les moyens de démonstration et de validation traditionnels mais aussi les conceptions architecturales des systèmes pour mieux les mettre à l'abri de défauts susceptibles de "passer au travers" tels que : défaut de conception, spécification incomplète, panne non catalectique, combinaison de panne élémentaire, point commun pour les alimentations et les masses, etc...

On voit que ces défauts peuvent apparaître tout au long du processus de définition et de réalisation d'un système. En particulier au niveau de la conception et des spécifications, c'est là un point particulièrement important, non seulement parce qu'il est difficile de détecter un défaut à ce niveau, mais aussi parce que les systèmes à venir seront de plus en plus complexes et à caractère multi-disciplinaire. C'est tout à fait le cas du Contrôle Actif Généralisé qui fait appel à l'Aérodynamique stationnaire et instationnaire, à la structure rigide et souple, aux qualités de vol, aux systèmes, etc...et pour lequel l'établissement de spécifications risque de dépasser la compétence d'un ingénieur système aussi qualifié soit-il.

L'introduction du calcul numérique compte tenu de l'expérience acquise sur ce système de commande vol électrique et aussi de celle accumulée depuis, ne modifie pas essentiellement la nature des problèmes rencontrés du point de vue sécurité. L'établissement de spécifications "sans faute" reste notamment un des problèmes à résoudre. On peut cependant noter que si les problèmes au niveau des "circuits électroniques" demeurent, on peut espérer mieux les résoudre grâce à une standardisation qui reste à mettre en place. En effet, de par la nature du calcul numérique, par opposition au calcul analogique, qui fait appel à une unité centrale de calcul capable d'assurer n fonctions on diminue de façon notable le nombre et la diversité des circuits électroniques nécessaires d'où, il faut l'espérer, une meilleure maîtrise de ces derniers. En revanche, cette simplification du "matériel" se fait au détriment du logiciel sur lequel se trouve transférée une partie de la complexité du calcul analogique. Il importe donc de bien maîtriser ce logiciel et à cet égard d'observer des règles de l'art aussi fondamentales que la simplicité et la transparence de ce logiciel, règles qui ont présidé à l'expérimentation du système de commande de vol électrique sur CONCORDE.

DESIGN AND DEVELOPMENT OF SOFTWARE FOR SEA HARRIER HUDWAC

by

E.P. Jones, Group Leader
S. Howison, Project Engineer - Sea Harrier Airborne Software
Software Design Group
Electronic Displays Branch
Smiths Industries Aerospace & Defence Systems Company
Cheltenham Division
Bishops Cleeve
Cheltenham
Gloucestershire

SUMMARY

This paper discusses the experience and approach of Smiths Industries Aerospace & Defence Systems Company in developing the airborne software for the Sea Harrier Head-Up Display and Weapon Aiming Computer System. This development activity was first begun in 1975 and continues today. The first Sea Harrier entered service with the Royal Navy in June 1979.

1. INTRODUCTION

The combination of two important pieces of military avionic equipment in recent years, namely the Head-Up Display Computer (HUD) and the Weapon Aiming Computer (WAC), resulting in the HUDWAC has meant the emergence of a powerful airborne computing system capable of providing in a single line replaceable unit the central computing and display generation for the Sea Harrier single seater VSTOL combat aircraft, which has recently entered service with the Royal Navy.

This paper will deal with the software design and development experience of Smiths Industries Aerospace & Defence Systems Company who are the prime contractors for the Sea Harrier HUDWAC system.

Figure 1a shows the relationship of the HUDWAC to the other avionic systems aboard the aircraft and provides an indication of the data flowing between units. It will be apparent that this organisation will impose a heavy burden on the HUDWAC and that the software system must be designed to ensure maximum throughput with minimum response delay to external events, and maximum fault tolerance.

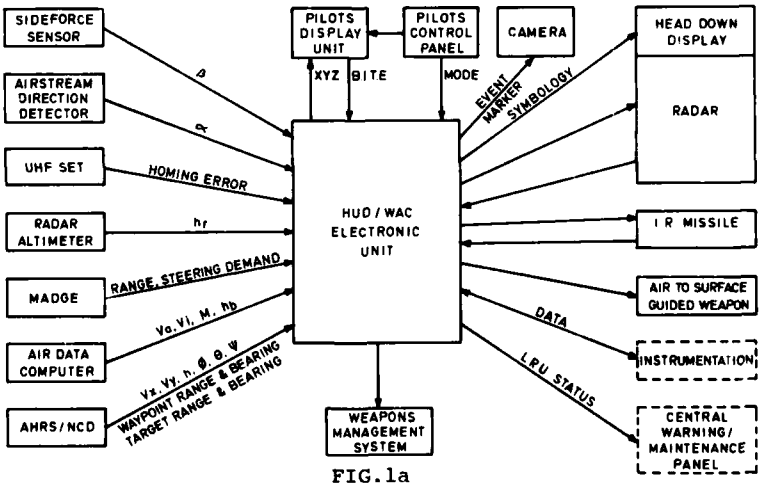


FIG.1a

In order to avoid any misunderstanding of terms with regard to the subject of airborne electronic displays, the following section defines the terms as used in this paper.

The Head-Up Display (HUD) system presents to the pilot in his forward field of view superimposed on the outside world, a collection of symbols, each representing one piece of either flight information or weapon aiming/attack information. The collection of symbols required at any instant, and their precise shape and size etc. are controlled by the HUD computer, which in the Sea Harrier is integrated with the weapon aiming computer in the HUDWAC Electronic Unit (HUDWAC EU).

The presentation of the symbology to the pilot is via a cathode ray tube and optical lens system in the Pilots Display Unit (PDU) which is mounted directly in front of the pilot in line with his forward field of view - see Figure 1b. The display is generated cursively (i.e. as a series of strokes) to obtain a high brightness capability and prevent obscuration of the pilots forward field of view.

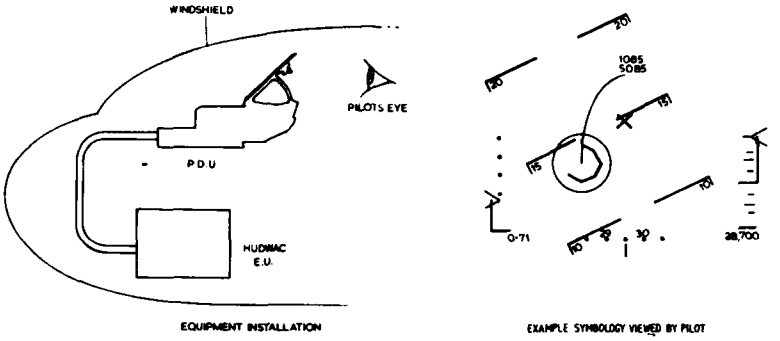


FIG.1b

The Head-Down Display generator (HDD) again generates symbolic flight information and is also integrated into the HUDWAC EU. The display format here is 625 line raster. This is presented to the pilot on the t.v. type radar display unit having first been video-mixed with the radar presentation, see Figure 1c.

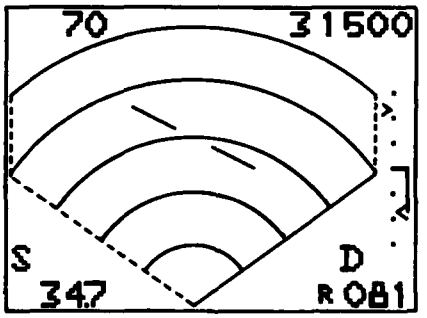


FIG.1c

2. SYSTEM OBJECTIVE

The HUDWAC system must be capable of providing the following general facilities, many of them simultaneously.

(a) Symbology generation for both Head-Up and Head-Down Display systems.

(b) Weapon aiming computations for a large variety of both air-to-air and air-to-ground weapons with manual or automatic release as appropriate.

(c) Flight path guidance for both target interception and aircraft to ship recovery.

(d) Pointing and control commands for Radar, Infra-Red missiles, and Air-to-Surface Guided weapons.

(e) First line avionic system test and fault indentification.

(f) Extensive operational self test and failure tolerance, accompanied by a graceful degradation of facilities, according to the severity of the fault.

Figure 2a shows how the foregoing objectives have been split into system tasks for subsequent implementation in software. The tasks have been grouped into facilities which relate to aspects of the system objective with the exception of the Executive and Control facility, which is an overhead necessary to manage the systems resources, and control the execution of the remaining facilities.

There are real time criteria in the foregoing list of objectives which are important, since failure to meet these criteria would result in a fundamentally unusable system.

Firstly, there is the problem of picture frame refresh on the two displays. The pilots display unit c.r.t. surface must be continually written to, and completely refreshed every 20 ms, so as to avoid flicker. The Head-Down Display generator also requires updating with new information every 20 ms.

The weapon aiming computations must be cycled sufficiently frequently to provide new solutions at a rate consistent with the rates of change encountered operationally. These cycle times vary but typically lie between 20 and 40 ms. The question of adequate computing cycle time is a general one and as well as weapon aiming computation, it applies equally to all other system objectives.

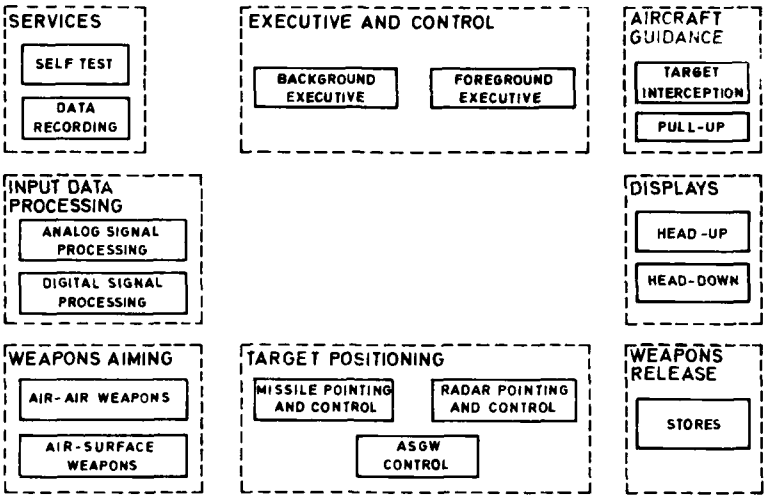


FIG.2a

3. HARDWARE CONFIGURATION

Having outlined the tasks required to be performed by the system we must now consider the characteristics of the hardware.

A block diagram of the hardware is shown in Figure 3a. The heart of the system is a well proven 16-bit main computer equipped with both DMA (Direct Memory Access) and program controlled data input/output, a fourteen level priority vectored interrupt system, an addressing range of 64K words, a core store for ease of reprogramming, and two real time references. Interfaced to the main computer but still within the same physical case are a 12-bit HUD processor, a HDD generator, an infra-red missile controller, and a host of external interfacing electronics (serial data transmitter/receivers, analogue to digital converters, digital to analogue converters etc.), through which communications are made with other pieces of avionic equipment as shown in Figure 1a.

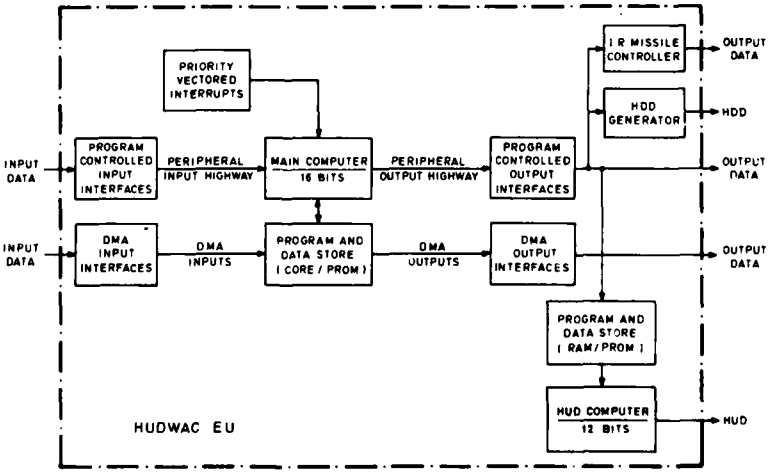


FIG.3a

The storage medium in the Sea Harrier HUDWAC is a combination of reprogrammable core and PROM. The current capacity of each is 16K words and 8K words respectively, with the possibility of increasing the core size to 32K should this become necessary because of the introduction of extra facilities. The PROM memory is used for standard programs which are unlikely to change, while the core is used for development programs and future enhancements to existing flight proven programs. The importance of a reprogrammable memory during development, and also throughout service life for certain key systems cannot be overstated. Although possibly more expensive at the outset, it pays for itself many times over by easing the embodiment of enhancements and modifications, which inevitably occur with great regularity even in the best managed projects.

The HUD computer is a 12-bit general purpose computer equipped with extra facilities for the generation of cursive graphics symbology. Such extra facilities include instructions to write lines, write strokes, write circles and circular sections etc.

The HDD generator has the capability to display raster symbology when so commanded from the main computer. The picture update information is transmitted from the main computer during the frame flyback period.

4. SOFTWARE DESIGN

4.1 System Requirements

Detailed statements of requirements are prepared in order to completely specify the functional requirements of the final system. Needless to say, this involves collaboration with the end user to ensure that the system will provide what is required, and also to agree any proposals for additional requirements as and when they arise.

The statements of requirements will identify the modes of operation of the system. A mode of operation is a unique arrangement of internal resources to provide a particular facility to the pilot. For example, the LAUNCH mode provides the pilot with all the information required to take off from the deck of a ship. When this mode is selected, as with any other mode, the HUDWAC is dedicated to the task at hand.

The Sea Harrier HUDWAC is currently equipped with 20 major modes of operation and 60 sub-modes of operation (a sub-mode being a mode within a major mode).

4.2 Design Specification - Tasks

As indicated earlier, the overall system requirements are split into a set of functional tasks, see Figure 2a. These tasks closely relate to the user view of the system, e.g. Air-to-Air Weapon Aiming.

The requirements of each task are specified in detail in the form of written description, mathematical relationships, flow charts etc., and form a detailed subset of the overall specification.

The interfaces between tasks are then defined in terms of data flow, which is effected via a common data area. This is an area of store in which dedicated labelled locations are used to hold information which is written to within a given task and read from within other tasks.

The interconnection of tasks for a given mode is shown in Figure 4a.

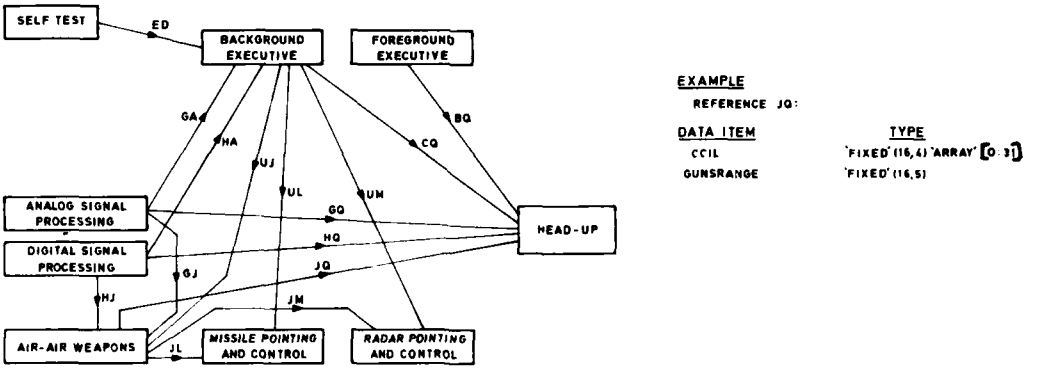


FIG.4a TASK INTERCONNECTION - AIR/AIR GUNS MODE

There is no flow of control between tasks. All tasks are entered from the Executive program to which return is made on completion of the task.

4.3 Design Specification - Modules

Each task can now be broken down into modules.

For the purposes of this paper, a module is a single program consisting of one or more procedures which is specified, written, and initially tested in isolation from other modules. It contains within itself sufficient software to test itself which is isolated from compilation when the module is proven (see Para. 6).

The function and input/output interface of each module can be specified together with guidelines for how it should be tested.

In this way, each module is specified, programmed and initially developed.

4.4 Executive and Control Program

This program exists to control the execution of the system tasks and to isolate the program modules from the problem of handling interrupts. See Figure 4b.

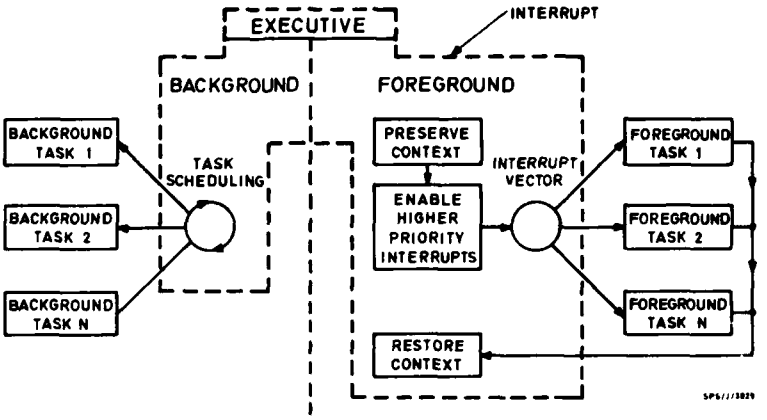


FIG.4b

The Executive is split into Background Executive and Foreground Executive. Background is the term used to mean the non-interrupted state, and foreground is the interrupted state.

4.4.1 Background Executive

The Background Executive is assigned the responsibility of establishing the required operating mode and controlling the execution of the various background tasks in order that the required objective is accomplished. The integer data word MODE is set to a unique value to indicate to any program module the operating mode required. Tasks are scheduled in priority order by procedure call, with parameters if necessary, when the Background Executive establishes that a particular task must run. The conditions for a task to run can be summarised as follows:-

The task concerned must:

- belong to that family of tasks pertaining to the current operating mode.
- be the highest priority task available for execution. A background task is available for execution if the time since it was last executed equals or exceeds its required cycle time.

Left uninterrupted the Background Executive would continually cycle, sampling the input conditions, establishing the required mode and scheduling the appropriate background tasks.

4.4.2 Foreground Executive

The Foreground Executive is entered wherever an interrupt occurs.

Figure 4c shows the available levels of interrupt in priority order.

0	INITIALISE
1	OVERFLOW
2	LOAD P
3	LOAD MANUAL
4	WATCHDOG TIMER
5	REJECT
6	REAL TIME CLOCK
7	HDD GENERATOR
10	HUD COMPUTER
11	INSTRUMENTATION
12	SPARE
13	MAG TAPE LOAD
14	GO
15	NO GO

FIG.4c

Certain of the systems tasks must be activated as a result of interrupt, these are foreground tasks. By way of an example of a foreground task, consider the set of modules we term HUD drivers. These are modules of software dedicated to the task of providing the HUD computer with fresh data for the currently selected set of symbols. They must be activated when the HUD computer has just completed a single frame of picture generation. This frequency of activation is required only for those symbols which display their information in an analogue form, and is necessary so as to present to the pilot the latest values of the selected parameters for both flight safety and picture quality reasons. If the refresh rate were too slow the display would appear 'steppy' and could provide a flight hazard if, for instance, the attitude information was 'old' when taking off or landing on the deck of a ship in the dark. The receipt of a HUD computer frame refresh interrupt must therefore cause the main computer to leave its current task and enter the Foreground Executive. The very fact that the interrupt has been accepted means that the task occupying the computer at the instant of the interrupt was either a background task or a foreground task related to a lower priority interrupt level. This is so, since before leaving the Foreground Executive to execute a foreground task, only those interrupts of a higher priority than the current one are enabled. In this way, the required HUD drivers are activated, and they produce the data required by the HUD computer during the next frame of picture generation. This principle applies to all the interrupt levels - each has a foreground task which is activated on receipt of an appropriate interrupt.

Figure 4d shows an example of the programs involved in generating a single HUD symbol.

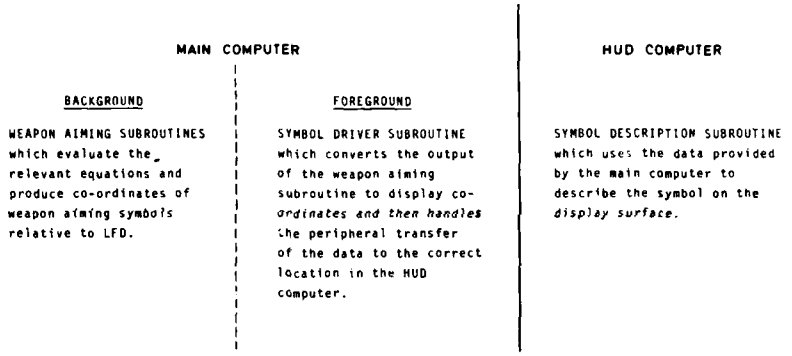


FIG.4d

The job of the Foreground Executive can be summarised as follows:-

On receipt of an interrupt:

- Preserve the current context, i.e. save all the information necessary to restart the interrupted program at the point of interrupt.
- Resolve any contention for processor time. This can occur if the two displays require servicing simultaneously since each must be served within a given time window. The window for the HUD computer is the time to perform its internal built-in test (BIT) facility and is approximately 2½ ms from interrupt. The window for the HDD generator is the frame flyback period of the raster waveform and is approximately 1½ ms from interrupt.
- Mask out all unwanted interrupts and enable the interrupt system.
- Enter the required foreground task.

On return from a foreground task:

- Re-establish the original context and return to the point of interrupt.

4.5 HUD Computer Program

As explained earlier, the HUD computer is a separate programmable computer within the HUDWAC EU. Its program resides in semi-conductor RAM store which is loaded from the main computer at power switch on. The program is structured as a set of symbol description sub-routines, the execution of which is decided by control bits sent from the main computer. Each symbol description sub-routine will specify on the display surface the precise form of the particular symbol it controls, also its position, size, brightness, etc.

4.6 Built-in Test Program (BIT)

This program spans both the main computer and its peripherals which includes the HUD computer and the HDD generator. The intention is to check as fully as possible the correct functioning of the system, and to prevent incorrect data being displayed to the pilot.

The integrity of the program is checked by summing the contents of each location and comparing the result with the known correct sum. This check is carried out in both the Main and HUD computers. If a HUD computer sumcheck fail occurs then it can signal the Main computer to reload its store. A main computer sumcheck fail will result in both displays being occulted.

The interfaces are checked by closing the loop between outputs and inputs allowing the same data to be output and then input and checked.

The displays are checked by built in circuitry controlled and monitored by software.

5. SOFTWARE WRITING

It was a requirement of the contract to use Coral programming language. This is a high level programming language. This not only made easier the task of writing the software, it is also much easier for someone unfamiliar with the system to understand the program. Coral has a block structure which combined with the easy to understand language makes the implementation of changes simpler. It also offers advantages in terms of efficiency of conversion to machine code. The main requirement of a real time language is speed of execution of the compiled code and compactness of the program. A Coral program is in general not more than 25% bigger than its hand coded equivalent.

All arithmetic is fixed point arithmetic as the Main computer has no floating point hardware and to use floating point software routines would incur too large a run time penalty. This limitation does mean that considerable care has to be taken in programming arithmetic operations to avoid overflows due to under scaling and loss of accuracy due to over scaling.

Tne HUD processor has its own independent program. This contains the software to drive the display system, including special sub-routines for non-standard display symbols. It also has software to carry out scaling, arithmetic and logical functions associated with supplying the HUD. All this software is written in Assembler language as it needs to be closely related to the output system hardware.

The HDD processor is also an independent processor. Its internal program is hardwired, but it is controlled by the Main computer. The Main computer software which controls the HDD processor is written mainly in Coral with only the interfacing to the HDD written in Assembler.

A host computer (HP 1000) is used for all Coral compiling and Assembling operations. All current programs are stored in this computer on magnetic discs. Changes are made using a text editor under on-line VDU control, by the programmer.

The Coral cross compiler is then used, either in batch running or in real time under on-line VDU control. This translates the Coral source program to assembler source which is then also stored on the host computer.

The Assembler is then used to convert this and the assembler source of any other program required to operate with it into binary code for the object machine. This assembler operation is carried out either by operator control from a VDU or by reference to a previously set up control file held on the computer. The binary source is stored on the computer. An assembler listing including store usage information and an error listing is produced by the Assembler on the computer line printer.

Originally the binary source was transferred from the host computer using paper tape as the transfer medium. This has now been replaced by a direct electrical link. This allows the object machine to be directly loaded from the host computer.

6. TESTING PHILOSOPHY

The system software is split into individual modules. Each of these modules is designed to allow it to be tested as an individual module prior to including it within the total airborne system. To allow testing, additional program is required. This extra program is permanently included within the module but by including this extra program within a Coral macro the extra program can be compiled or ignored according to the macro definition. Thus by just changing one line of program, the macro definition, the extra program can either be compiled into assembler source or ignored. This facility allows the testing of modules without seriously changing them from their intended system state.

As an example of this a simple Coral program to add two numbers together is shown. Both the numbers, NUMA and NUMB, and the resulting answer, ANS, are defined as Common variables which are assumed to be used in other modules. The program therefore needs additional software to test this module on its own. This extra software must supply values for NUMA and NUMB and communicate the results to the operator by some means, in this case by sending them to a printer.

```
"SAMPLE PROGRAM  ISSUE 1  AUTHOR S HOWISON  21.8.79"
'COMMENT'  This is an example of the use of the DEBUG macro to test modules;
'DEFINE' DEBUG(D) "D";
'COMMON' ('INTEGER' NUMA, NUMB, ANS);
DEBUG ('COMMON' ('PROCEDURE' PRINT ('VALUE' 'INTEGER');
                 'PROCEDURE' NEWLINE));
'BEGIN'
DEBUG ('INTEGER' N;
'INTEGER' NUMA;
'INTEGER' NUMB;
'INTEGER' ANS;)
'PROCEDURE' ADDNOS;
'BEGIN'
ANS: = NUMA + NUMB;
'END';
DEBUG ('FOR' N: = 1 'STEP' 1 'UNTIL' 20 'DO'
'BEGIN'
NUMA: = N;
NUMB: = 'IF' N<8 'THEN' 4 'ELSE' 56;
ADDNOS;
PRINT(NUMA);
PRINT(NUMB);
PRINT(ANS);
NEWLINE
'END';
STOP: 'GOTO' STOP);
'END';
'FINISH' OF SAMPLE PROGRAM
```

In the example the third line defines whether the extra program needed for testing is to be compiled. As it is, it defines the contents of DEBUG brackets as to be compiled. If it is changed to 'DEFINE' DEBUG(D)" "; then anything inside a bracket which is preceeded by DEBUG will be ignored. Thus the common declaration of the library routines Print and Newline will be ignored, as will the declaration of the variable N and the whole of the final block.

When the DEBUG is defined as to be compiled the final block is compiled. This block will set values into NUMA and NUMB, it will then call the procedure ADDNOS and then will print the result of this procedure. The results are printed alongside the input values NUMA and NUMB. The final block carries out twenty consecutive tests on procedure ADDNOS, stepping NUMA from one to twenty and assigning NUMB two different values, four for the first seven tests and fifty six for the rest of the tests. Thus the first value of ANS printed is five, the second is six and the last seventy six.

It can be seen that although the program is changed by the state of the macro the only procedure in this module, ADDNOS, is tested without actually changing it.

When testing a module all the software in the module is compiled, with the macro definition for testing enabled. It is then assembled with any other software required, such as library routines. The resulting binary code is loaded into a special Sea Harrier Software Module Testing Facility, See Figure 6a. This consists of the basic Sea Harrier Main Computer, a large core store, a basic set of test equipment, a paper tape reader, a VDU and a printer.

The binary code is loaded from the host computer. The module software can then be operated either directly or via a Monitor software package controlled from the VDU. Listings of the inputs and outputs the module is generating are output to the printer. These results can then be checked. They are either checked against tables of expected results, or by manual checking, or by test software written either in Fortran or Basic and run on the host computer.

As stated earlier the arithmetic operations are all carried out at fixed point scaling and one of the main functions of module testing is to check that the scalings have all been correctly chosen for the full range of inputs.

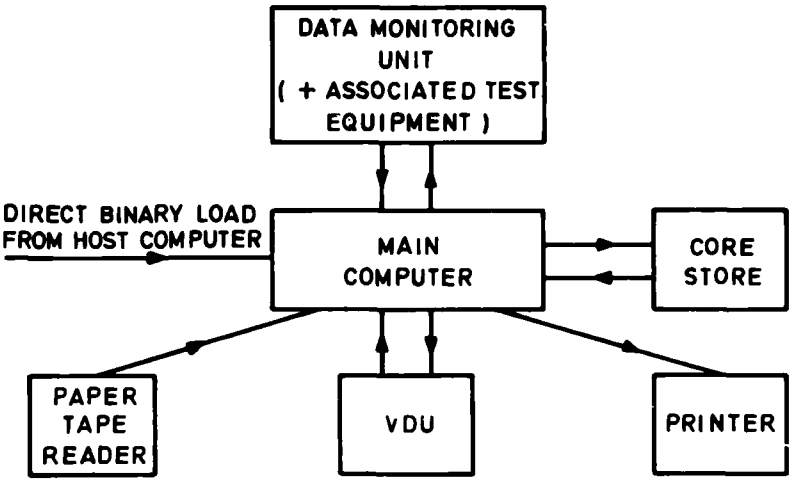


FIG.6a

After module testing is successfully completed the macro definition for testing is disabled and the module re-compiled. It is then assembled with all other modules of the airborne system to produce the binary code for the full system. This is then loaded into a Sea Harrier unit. This unit is part of a Sea Harrier Systems Test Station, See Figure 6b.

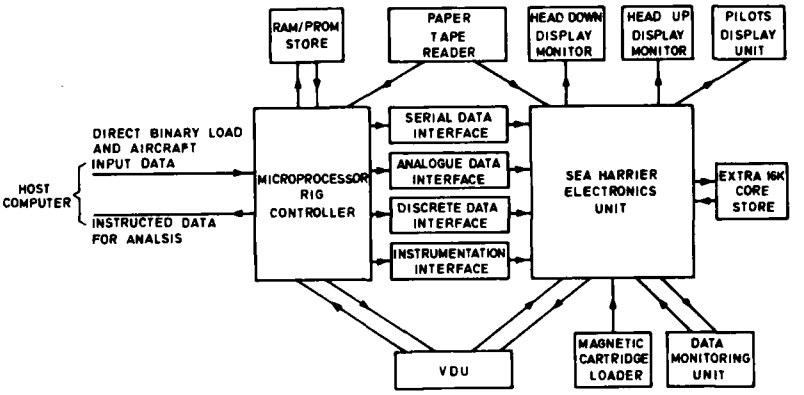


FIG.6b

Using this station the system is commissioned. This is carried out by careful checking of the software's operation and correction of any errors. The procedure for making changes is to first change the Coral source using the Text Editor on the host computer. Then to re-compile the affected module with the macro for testing enabled. The module is then assembled and retested on the Module Test Facility. The extra testing part of the module would have been edited as well, so that it tests that the change made operates correctly. The macro is then edited, to remove the module testing, and the module re-compiled. The system is then reassembled and the binary code is transferred from the host computer to the Sea Harrier System Test Station. The system is then checked to ensure the change is correct.

Once a software system is thought to be fully working it is formally tested. It is tested using a formal test specification. The test specification checks that correct results are obtained for various input cases. For example, if Radar Height Invalid is set on the appropriate input, Radar Height is not displayed. Also if a Heading of twenty degrees is input, the HUD and the HDD both display Heading at twenty degrees. Errors found at any stage during this testing not only have to be corrected by re-compiling and reassembling as said before, but also testing has to be restarted from the beginning of the test specification to be sure no other errors have been caused by the change.

After formal Static testing has been successfully completed Dynamic testing is carried out. This is done using the Sea Harrier Software Test Station and the host computer. The host computer is used to supply the Test Station with the data it would normally receive in the aircraft during flight. The host computer generates this from a software model of the aircraft which is operated through a defined set of flight profiles. The Sea Harrier processor outputs data to the host computer to be stored and used later for analysis. To carry out this analysis the host computer runs a model of the HUDWAC function being tested. It compares the results from this with the data it received from the HUDWAC. The

results of this analysis are output to the host computer's printer and its plotter. These results are then available should any problems be found during development or service use. The profiles are chosen to test the HUDWAC function under scrutiny to the limits of its operational envelope. An example of this is in testing the return to ship mode. For this a set of profiles are defined covering approach to the ship from many different directions, at different speeds, different heights and varying all the start conditions that might affect the operation of this mode. The dynamic testing then checks the operation from these start conditions to the end point of landing on the ship for each case.

The dynamic testing with the HUDWAC being driven by inputs from the host computer is carried out in real time. The analysis of this dynamic testing is carried out afterwards as a batch process.

The software system contains within itself certain software to test the hardware of the HUDWAC. This software carries out tests on the input system, the main processor and the HUD and HDD processors. If a fault is detected the software will act on it and not use the data or the hardware that has been identified as faulty. Thus only information and hardware not known to be faulty is used.

7. SOFTWARE DOCUMENTATION

Software documentation is split into two parts, Development documentation and Customer documentation.

7.1 Development Documentation

Development documentation is used to define what software is needed and what software has been written. The software required is firstly defined by an SOR, Statement of Requirements. Separate SOR's define the HUDWAC Symbology and Inputs, the Weapon Aiming equations, the Processor Hardware and the Testing Requirements (BITE).

Changes to the software are split up into amendments to the existing system. These amendments, identified by numbers, list all the modules affected by the change. Each module has a History File. This History File contains a copy of each standard of the module, plus the printer output from the module testing of that standard of the module. It also contains any other information relevant to testing that module i.e. Fortran checking program results. Also in the file is a History sheet, this lists which standard of the module was included in which standard of the system and what the changes between different standards of the module are, who the author of the change is, the date the change was made, with the number of the amendment which defined the change.

As an aid to documentation and to development design a special suite of programs were produced. These allow a 'database' of the source programs in a system to be generated. This database can then be interrogated to see how that system operates. For example, where a variable is declared its scaling can be found, or all calls from a certain procedure can be listed.

In the following example all the references to Common Variable VT and all the Procedures called from Procedure Backgexec are listed. Against the variable VT is listed the access mode, R for Read From, W for Written To, and P for Procedure actual Parameter. The commands input by the operator are shown between brackets [ ]

```
[RU,COPAC,45,,SEAHAR2C,,SH]
COPAC/03    File : Sea Harrier HUDCASS 2c  Date : 5.1.1978
System Number : 1X46SSP3
Date of System Generation : 4.9.1979

[REFTOVAR.VT]
Procedure/Program references to Common Variable : VT
Procedure INSTRUMENT     Task Ref I  Access mode    R
Procedure ANALOGPROC     Task Ref G  Access mode     W
Procedure AAJGGUNS       Task Ref J  Access mode    R
Procedure DENSITYRATIO   Task Ref F  Access mode    R
Procedure AIRDENSITY     Task Ref F  Access mode    R
Procedure YELYECT        Task Ref F  Access mode      P
Procedure TASWIND        Task Ref F  Access mode    R P

[CBPROC.BACKGEXEC]
Procedure called by Procedure : BACKGEXEC
BACKGCORE
INSTRUMENTL
FAULTS3
ANALOGPROC
DIGITALPROC
FAULTS2
INPUTB1TE
FAULTS1
INDISC
CPUBITE
WATCHDOG
OUTMASK

[END]
```

The facility is operated in Real Time on the host computer, output can either be to the users VDU or to the printer if a permanent record of the inquiry is required. The database is generated by compiling the programs with the Coral compiler set to documentation output. In this state no assembler source is produced by the compiler. Instead documentation object data is produced. A loader program checks this object data for errors and then loads it into the database. This table can then be accessed and interrogated by the user as described above. While all the information available from this database can be found by looking through the source code listings, considerable time is saved by having this facility on a system of this size and complexity. All the software which writes to the database requires special passwords to stop the database being accidentally corrupted.

The final part of the development documentation is the test specification. This defines the testing for a standard of system software. Each amendment requires a change to test specification to test the change.

7.2 Customer Documentation

This is carried out in accordance with AVP70 SPEC.4.

Customer documentation is split into four levels of documentation. At the first level an overall description of the system is given including the basic hardware system. At this level the software is split into separate main tasks. At the next three levels the software is described one task at a time. The documentation is physically split into volumes, thus one volume has the level 1 description, the next volume the levels 2, 3 and 4 for a particular task, the next volume again the levels 2, 3 and 4 for the next task. This method of splitting the documentation has allowed the documentation to be written in parallel with the development of the software; once an area of software is complete the volume covering that task can be issued.

The level 2 documentation contains a general description of the function of the task. The level 3 documentation covers the task in far greater detail. It includes flow diagrams of control and data flow. The level 4 gives the source program listings of the modules within the task.

8. SOFTWARE PACKAGE PLANNING

Due to the huge size of the total task it is not feasible from our or the customer's point of view to deliver all the software at one time. It is therefore necessary to split the software into packages consisting of different standards and sub-standards. It is also necessary as the project progresses to update and alter these to suit the customer's delivery and testing needs and in co-ordination with the availability of other manufacturers avionic equipment. There is also a need for the specifications and documentation to keep in step with the software standards. This is done by a committee consisting of the company, the customer and the airframe manufacturer.

The total system software is split into standards to suit the customer's delivery requirements for the aircraft. These main standards are identified by a number. Then the standards are split into sub-standards to suit the flight testing program. These sub-standards are identified by a letter suffix to the standard number identifier. Once a month the committee considers the software in terms of the progress of work on new software, the state of the flight program and any changes in the priority for software facilities. The definitions of the standards and sub-standards are then updated along with their delivery dates. The requirements for documentation of the sub-standards are then discussed. Normally each sub-standard has all the facilities of the previous sub-standard plus one or more new facility. Each standard normally continues from where the previous standard stopped. Each completed standard is a fully cleared software system available for service use.

9. CONFIGURATION CONTROL

The standard of each software module is identified by an issue number. This issue number is listed at the head of the module alongside the title and an identifier code. Once the standard of a module is fixed, it has been tested within a working system, it is necessary to store that standard on a more reliable medium other than a computer magnetic disc. It is stored as a paper tape in a software library. With the paper tape is stored a history sheet listing the originator of the module, its title, the date it was generated and the date it was filed in the library. In addition, a copy of the paper tape is stored in a secure store in case the library tape is lost. If during development that standard of that module is required a copy can be obtained from the library. Binary tapes of systems are also held by the library in the same manner. The library is also responsible for generating Magnetic Cartridges from paper tapes. When a software item is required for delivery to a customer the library generate it. It is then checked by Quality Assurance Inspection before being delivered.

10. FLIGHT TRIALS SUPPORT

The Sea Harrier HUDWAC provides a large percentage of the operational capability of the aeroplane, and as such is a key item in the programme of flight trials. This necessitates a separate software support team based at the flight trials site.

The responsibilities of this team are to provide the customer with fully developed modifications to enable the trials to proceed according to plan. Such modifications arise for a variety of reasons some of which are discussed below.

Trials modifications may be requested by the customer to explore the potential provided by certain facilities which have come to light since the specification for the equipment was agreed. Such trials modifications once specified and agreed, will be integrated into the flight trial programme by a specified date. Their future will depend on the outcome of subsequent flight trials but can vary from being discarded as unsuccessful to being fully embodied into future software standards.

Software implementation errors (bugs) which have escaped the testing net will occur from time to time. These must be corrected speedily and accurately.

In both the above cases and indeed in all cases resulting in a software modification at the flight trials site, it is vital that the necessary information is fed back to base for processing through Configuration Control.

The ability to support software development in the field depends on the team involved being equipped with a computer system equivalent to that used at base. In the case of the Sea Harrier HUDWAC, a DEC PDP 11 system is used and is host to the full suite of support software packages.

11. CONCLUSIONS

The approach taken to producing software as described in the preceding paragraphs has proved highly sucessful. It was possible to quickly produce an initial system for rig evaluation and to closely follow this with a sequence of software packages to correspond with the sequence of flight trials so as to arrive at the initial "in service" capability on schedule.

This project was the companies first experience of using a high level programming language for a production avionic system, where computing time and store space are at a premium. The experience generally, has been a good one. Using Coral 66 has enabled us to produce correct and intelligible programs more quickly than with assembler and this has proved an important bonus in the area of off site support where ease of understanding and speed of modification, coupled with clear communication back to base, has helped towards providing good customer support with only a few staff based off site.

Coral 66 is obviously not perfect but our experience would show it to be a major step in the right direction. Future languages will have to show significant advantages in order to replace Coral 66 in the type of project discussed in this paper.

The next major development is likely to be a formalised design and development methodology for real time systems along the lines of the current Mascot system. With this in mind SI are currently employing Mascot in a prototype ground based electronic displays project.

SOFTWARE FOR AN INTEGRATED FLIGHT CONTROL AND NIGHT VISION SYSTEM FOR MILITARY HELICOPTERS

P. Elzer, F. Figel, W. Hoffmann

Dornier AG
Postfach 1360
D-7990 Friedrichshafen
West Germany

ABSTRACT

The paper describes the functions and structure of an integrated digital flight control and night vision system for military helicopters. Its partially redundant components are connected via a serial bus. This resulted in a system with distributed processors. The software had to be structured according to the distributed character of the system and had to take into account the master-terminal principle of the bus architecture. It comprises the algorithms for flight control and handling of the night vision equipment as well as management functions for the bus, the displays, input commands and error handling. Development of the software and integration of the system were supported by appropriate hardware and software aids. A special device was developed to facilitate integration of the distributed system. For this the same modular electronic components were used as for the control system.

1. INTRODUCTION

When flying a helicopter at night and under adverse weather conditions the reduction of visibility has to be compensated by technical means. However, available equipment for night vision, like e.g. FLIR, LLLTV, Night-vision goggles, is not adequate to replace natural vision to an extent which would allow the pilot to fulfil all necessary missions. It is therefore necessary to support the pilot by providing better handling qualities of the helicopter and by an integrated display of flight control information and housekeeping data.

These requirements result in a functionally highly complex control system on board. It is no longer feasible to design it in such a way that just the necessary components and subsystems are added together. In order to reduce configurational complexity and cost of the system it is rather necessary to develop an integrated structure, within which sensors or signal processing units can be used simultaneously for several different purposes. So e.g. data from one particular sensor can be used for control purposes and display of flight variables. The success of this technology depends extremely strong on quality and structure of the software within the integrated system. Therefore special efforts have to be made to develop it in a well structured way and to test it to the largest possible extent, especially in the integration phase.

An experimental system to investigate the soundness and feasibility of these design principles is being built under contract with the German Federal Ministry of Defense. Parts of it have already been evaluated in flight test; the final form is currently being implemented.

2. ARCHITECTURE AND CAPABILITIES OF THE FLIGHT CONTROL SYSTEM

2.1 Structure of the System

The integrated system consists of

- pilot night vision equipment
- flight control functions
- integrated displays and controls.

The components of that system are located in different parts of the helicopter according to the requirements of the respective peripheral units (sensors, actuators, displays). The flight control system for the helicopter is designed as a fly-by-wire system and therefore has to be redundant for safety reasons. Only digital technology allows to build such a system with reasonable effort and in a sufficiently flexible way.

The work described in the paper is part of the experimental work for HSF IV (flight control of helicopters) which was conducted under Contract No. T/R 720 R/7600/43102 of the German Federal Ministry of Defense (BMVg)

In order to minimize the number of wires connecting the individual signal processing units and to achieve a standardized hardware interface, which additionally allows relatively simple extension and modification of the system, the serial bus according to MIL-STD-1553 is introduced.

Fig. 1 shows the structure of the hardware. For the redundant flight control functions three identical signal processing units U1, U2, U3 are necessary. Proper functioning of the units is checked by voting. Thus one hardware failure can be tolerated without affecting the control function. This is sufficient for an experimental system, where there is always a second pilot who can take over control in case of an emergency. An operational system must be able to tolerate at least two independent hardware failures, may be by additional self-check (BIT). The necessary sensor data as well as the pilot's control commands are input into the system threefold. In contrast, actuators are only duplicated, because the hydraulic system of original helicopter is only duplex as well. Therefore it is necessary in this case to check malfunctioning of the actuators by additional devices. Three additional signal processing units (U4, U5, U6), which are located in the cockpit, are used to process the pilot's commands, prepare the display output and connect the display and input units in the cockpit with the controlling units via the three serial busses.

The pilot night vision system (PNVS) consists of a TV-camera (which will be replaced by a FLIR in later phases of the project) mounted on a slewable platform in front of the nose of the helicopter. This platform is stabilized with respect to azimuth and elevation. The line of sight of the camera is locked to the pilot's head position by means of an angular head position detector (helmet mounted sighting system). The image, produced by the FLIR or TV, is displayed optionally on a panel mounted vertical situation display (VSD) or on a helmet mounted VSD. In both cases it is overlaid by symbols showing the state of the flight variables.

For post flight evaluation the important signals are recorded during flight tests on tape cassettes and simultaneously transmitted to a ground station via telemetry.

During integration of the system or the pre-flight tests in the helicopter a test support computer (ITP) can additionally be connected to the serial bus. Thus checks can be initiated and important data be documented as hard copy.

2.2 Operational Software

In order to achieve a well structured digital system and to facilitate its development, integration and modification, the structure of the software has to be rather rigid, and appropriate development and integration tools have to be provided.

The various software jobs can be classified as follows:

1 Management and control of the serial bus
2 Flight control algorithms
3 Control algorithms for line of sight of the camera
4 Display of flight variables (VSD)
5 Display of system status
6 Processing of pilot's commands
7 Detection of malfunctions and failures
8 Data transmission and recording.

For reasons of redundancy and speed of processing the various software jobs are handled in a decentralized fashion in the signal processing units U1 to U7. The technology of the MIL-Bus requires separation of 'master' and 'terminal' functions. The strategy chosen for redundancy normally requires that there is a permanent master for each bus (U1 for A, U2 for B and U3 for C). U4 to U7 are operated as terminals. Only in case of a failure in one of these units other (predefined) units on the respective bus can be mandated to act as masters in order to maintain the remaining functions. The test support unit (ITP) can become master on request if it has to be used for test of the system.

In the following sections the functions of the various software jobs will be described in more detail.

2.2.1 Management and Control of the Serial Bus

As mentioned above, each of the three bus lines is controlled by one permanent master. However, in cases of emergency or if the test support unit, the integration and test panel (ITP), has to be used, the role of 'master' can be assigned to another unit.

Data, which have to be transmitted via the bus, are organized in messages, which have a pre-defined format and can be identified unambiguously. It is possible to define an arbitrary number of such messages. A message contains the number of data-words, the source and the destination of the transmission. This allows identification of a message by the signal processing unit concerned and error-checking. Messages can be transmitted either periodically or on request. Terminal units are able to inform the respective master units that they want to transmit a message.

The serial bus is also used for mutual synchronization of the three redundant branches of the system.

2.2.2 Flight Control Algorithms

In order to achieve good flight characteristics and handling qualities of the controlled helicopter over the entire flight envelope, even in the case of strong disturbances, the control system has been designed as a so-called 'manoeuvre demand system' with high authority. Flight tests in the first phases of the experimental program showed that pilots preferred direct control over the rotational degrees of freedom of the helicopter. Therefore angular velocity, attitude, heading, and altitude of the helicopter are controlled.

The control laws have been derived from the aerodynamic characteristics of the helicopter. As a nucleus they contain a proportional-integral (PI) control algorithm for each of the four controlled axes (pitch, roll, collective, yaw). Additionally there are (partially nonlinear) functions for the processing of sensor data, adaptation of the flight dynamics to the entire flight envelope, compensation of disturbances and cross-coupling, and for transformations of control parameters which are necessary to maintain the orthogonality of the controlled axes.

A major advantage of this controller design is that adaptation of parameters is very easy when changing from simulation to flight tests. The reason is that each parameter can be interpreted as a physical entity.

Some sensors have only been duplicated. If one of these two breaks down, the control function, which depends on these data, is replaced by a simpler one (functional degradation). The respective functions have been designed in such a way that the reactions of the helicopter to the pilot's commands do not change much.

2.2.3 Control Algorithms for Line of Sight of the Camera

The platform, on which the night vision camera is mounted, can be stabilized according to two different modes:

- Fixed position with respect to the ground
- Fixed position with respect to helicopter.

The pilot can change these modes in flight by a manual switch.

The control algorithms for both modes are basically PI-algorithms with feedback of some of the sensor data. Some additional nonlinear algorithms allow fast mechanical reset of the platform, or they take over control in case the limits of mechanical movement of the platform are reached.

2.2.4 Display of Flight Variables

The pilot is informed about all necessary flight variables (like e.g.: attitude, heading, altitude, velocity) in an integrated form by means of symbols which are superimposed on the image provided by the night vision system. The integrated image with symbols is displayed either on a panel mounted vertical situation display (VSD) or on a helmet mounted display (HMD). The symbols are generated by a separate device, the symbol generator. By means of appropriate software, resident in the signal processing unit U4, the sensor data, which have been preprocessed in units U1 to U3, are transformed into parameters for form and position of the symbols. In particular, this software has to perform the following main tasks:

- Filtering of sensor data
- Computation of the entities to be displayed
- Preparation of the various sets of symbols (for cruising, hoovering, HMD, VSD, etc.)
- Computation of position parameters for the symbols
- Output of computed parameters to the symbol generator.

2.2.5 Display of System Status

In addition to the VSD there is another display device in the cockpit, the multi-function-display (MFD), which e.g. can be used to display system status data or as a backup for the VSD. Fig. 2 shows a schematic view of the screen and control board of that device together with the kind of messages which typically would appear on the screen.

As far as system status information is concerned, the modes of the MFD are:

- General checklist
- Pre-flight test
- Input of parameters for control functions
- Input of parameters and selection of modes for VSD.

Additionally error messages can be displayed. They have a higher priority than the usual dialogue messages and are further emphasized by use of a 'blink mode', but they can be deleted again by a mode key.

The software for the MFD is resident in signal processing unit U5 and performs the following main tasks:

- Extraction of actual values out of the system
- Preparation of dialogue texts
- Formatting and display of values and texts
- Processing and display of error messages.

2.2.6 Processing of Pilot's Commands

As Fig. 2 shows, the keys of MFD are also used for input of pilot's commands. The software for the processing of these inputs is resident mainly in unit U6 and performs the following main functions:

- processing of each input (conversion of input into mode, function and value; transmission to the unit concerned)
- acknowledgement of each input on the MFD
- prevention of erroneous input, e.g. from simultaneous operation of two keys.

2.2.7 Detection of Malfunctioning and Failures

Prior to takeoff the redundant system has to be checked for proper function of all components. This implies a rather extensive test of each individual device. Sensors and actuators are stimulated (as far as possible), and the pilot is informed via the MFD which input the system expects from him, e.g. via switches, keys, stick, collective pitch, pedals. The signal processing units mutually check each other. Only after proper functioning of all devices has been verified, the helicopter is cleared for takeoff.

In flight all components of the control system are regularly checked by means of 'voting' (2 out of 3). If this indicates malfunctioning, the signals from the faulty component are ignored. A failure is defined as a malfunction over a certain predefined time period. After a failure has been recognized, the respective component is totally ignored by the signal processing system.

Each detected failure is indicated to the pilot by a redundant degradation indicator. Sensors, which are not triplicated, are monitored by additional software.

2.2.8 Data Transmission and Recording

In order to be able to monitor flight tests from the ground, all relevant signals are transmitted via telemetry to a ground station and recorded and evaluated there. These data are collected in unit U6 and transmitted sequentially.

For a computer-aided off-line evaluation of flight tests the telemetry data and some further signals are recorded on cassettes on board. This has the advantage that these data are free from transmission noise. They are also collected and output by unit U6.

3. DEVELOPMENT AND INTEGRATION TOOLS

3.1 Hardware Support

3.1.1 Support for Program Development

For the program development process proper, which is mainly characterized by the necessity to handle extensive texts, which can either be programs or specifications, conventional equipment can be used most readily. The main criteria here are speed of response and capacity of background storage media. Therefore an available PDP 11/70 was used to run the editors, cross-assemblers and other development support programs. Its peripheral equipment consisted of a large disk, magtape, line printer, tape punch and CRT-terminal(s).

3.1.2 Support for System Test and Integration

For test and integration of the final system, however, no adequate hardware support was available. The main problems in this area are to give the development engineer access to every line of the bus and every single bit of the program under test, to facilitate re-programming of PROMs if necessary, to facilitate access to the signal processing units, which are hidden somewhere in the system, to monitor data transmission on the bus lines, and to apply test techniques (like breakpoints etc.) without uncontrolled modifications of the original program.

Therefore a special device was developed, the 'Integration and Test Panel' (ITP). It basically consists of a small computer of the same type as the signal processing units U1 to U7 with fast random access memory of adequate size (64 K of 16 bit words) and all necessary indicators and input switches which enable the test engineer to work with the signal processing unit in the same way as if it were a conventional computer with an operator's console. It was constructed out of the same modular electronic components which were used for the control system. The ITP can further be equipped with a paper tape reader and punch, a CRT-terminal with keyboard, a PROM-programming unit, and a small printer. Fig. 3 shows a typical ITP configuration.

In principle it can be used in three modes:

- The built-in processor replaces one of the signal processing units in the system under test. The program in that unit can be checked out with the support tools available in the ITP.
- The ITP can be coupled to the serial busses and monitor the data flow and functions of the bus system.
- The ITP can simulate the system by producing test data and thereby allows to still further check out one of the signal processing units.

3.2 Software Technology and Tools

3.2.1 Development Principles for Application Software

All program modules were developed according to uniform standards which were established prior to the start of the project. They all had to be of the same structure. Program modules, which were to be used in several signal processing units, were produced once and copied into the units concerned. Data, which were to be used throughout the system, had to be named uniformly. Drivers for peripheral units, if necessary, were developed according to a general model and adapted to the individual requirements by minor changes. During design of the programs potential sources of errors were identified and exception handling routines provided there. These consist of counters which monitor the number of occurrences of the respective error for later investigation.

3.2.2 Software Tools for Test Purposes

Special support software was developed for use within the ITP. Its main purpose is to facilitate the test process by providing a comfortable user interface and at the same time to protect the software under test from destructive handling errors during the test process.

Some of its major functions are:

- Execute 'Program under Test (PUT)' in real time with real peripherals
- Execute PUT under approximated real time conditions with break points, at which the complete status of the PUT can be interrogated
- Execute PUT in step mode, again with complete program status available
- Modify program in RAM of ITP.

- Program and check PROMs directly from ITP
- Monitor the values of important variables, which can be selected by the test engineer, either in their digital representation or directly in analog form by means of meters driven by DACs.

All functions of the ITP, which imply interaction with its built in processor and/or memory, can be invoked and the resulting information displayed via an interactive CRT-terminal.

4. EXPERIENCE AND CONCLUSIONS

Generally speaking, the technology, which was used, and the tools, which were developed and used, proved successful.

In the flight tests to date a partial system has been used which does not yet have redundant components, but for all practical purposes implements the control functions of the final stage. Only collective control was obtained as in a conventional 1 : 1 control system. It turned out that the requirements were fulfilled to a very large extent. The helicopter showed high precision of attitude and heading control. The number of necessary control activities by the pilot could be drastically reduced. Pilots proposed an automatic altitude control, which consequently was integrated into the system.

As far as the electronic equipment was concerned, it also proved successful. Especially the principle of modular, self-contained subsystems proved very useful under reliability aspects.

The test functions of the ITP turned out to be very useful for speeding up the entire integration process and for monitoring program interfaces during execution of the program system. Especially helpful were the functions for checking and re-programming PROMs. Coding of PROMs was at least three times faster than with conventional methods, while at the same time less handling errors occurred. A single function: 'comparison of PROMs with paper tapes', resulted in the detection of faulty PROMs as well as the identification of defects in printed circuits.

The error handling by counters helped in detecting transient and sporadic errors and supported the indirect detection of errors by deduction.

However, some extensions of the tools would have helped in further improving their effectiveness. So, e.g., the implementation of monitoring functions and checkpoints in hardware would allow to test the programs under realtime conditions in nearly all cases, which seems to be of extreme importance for the kind of systems we are dealing with here.

5. REFERENCES

[1] A Tri-Service Military Standard for Aircraft Internal Time Division Command/ Response Multiplex Data Bus. MIL-STD-1553B, May 1978

[2] H.J. Bangen, W. Hoffmann, W. Metzdorff:
Implementation of Flight Control in an Integrated Guidance and Control System;
AGARD Conference Proceedings No. 258, Oct. 1978, pp. 24-1 to 24-11

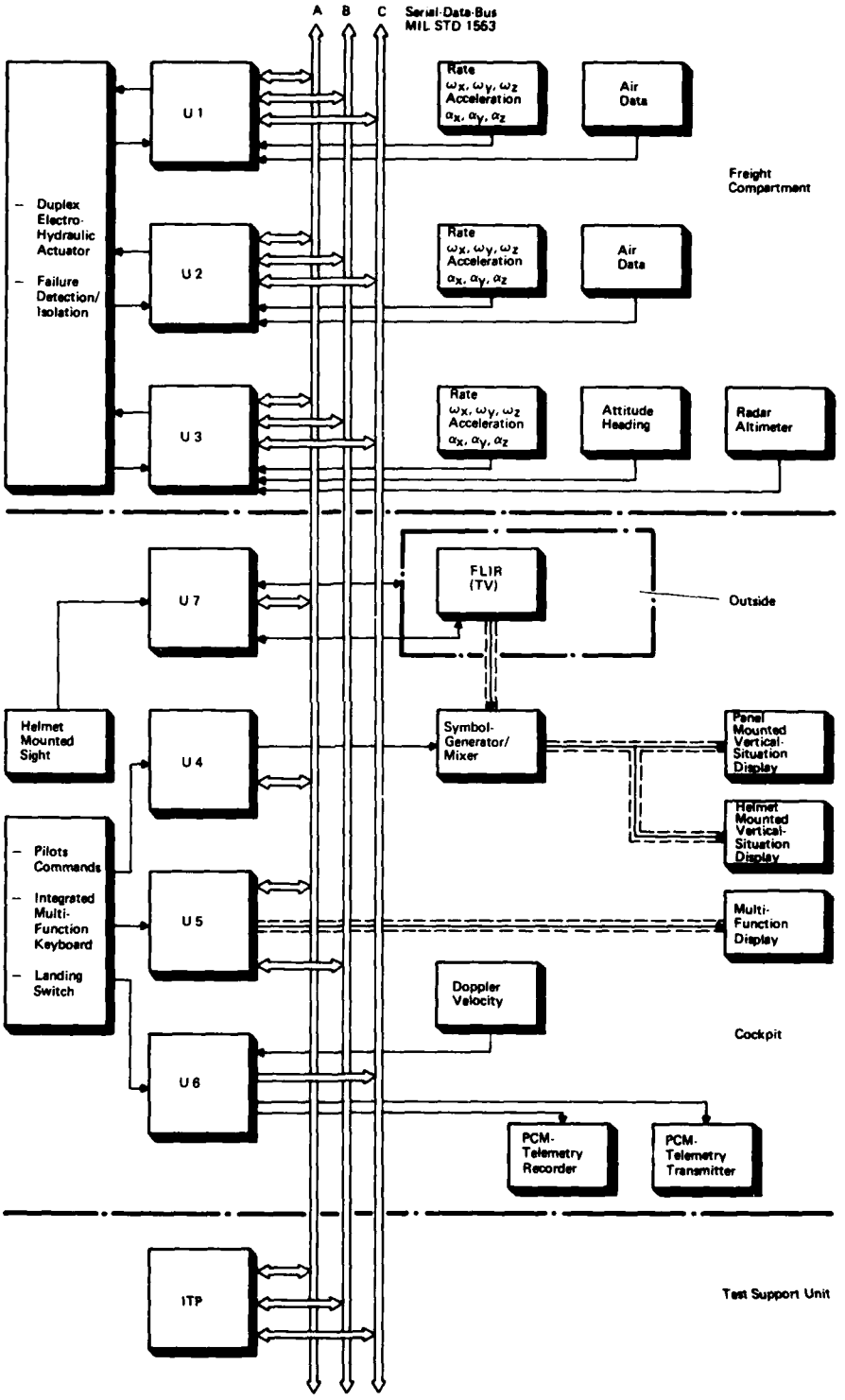


FIG. 1: HARDWARE STRUCTURE

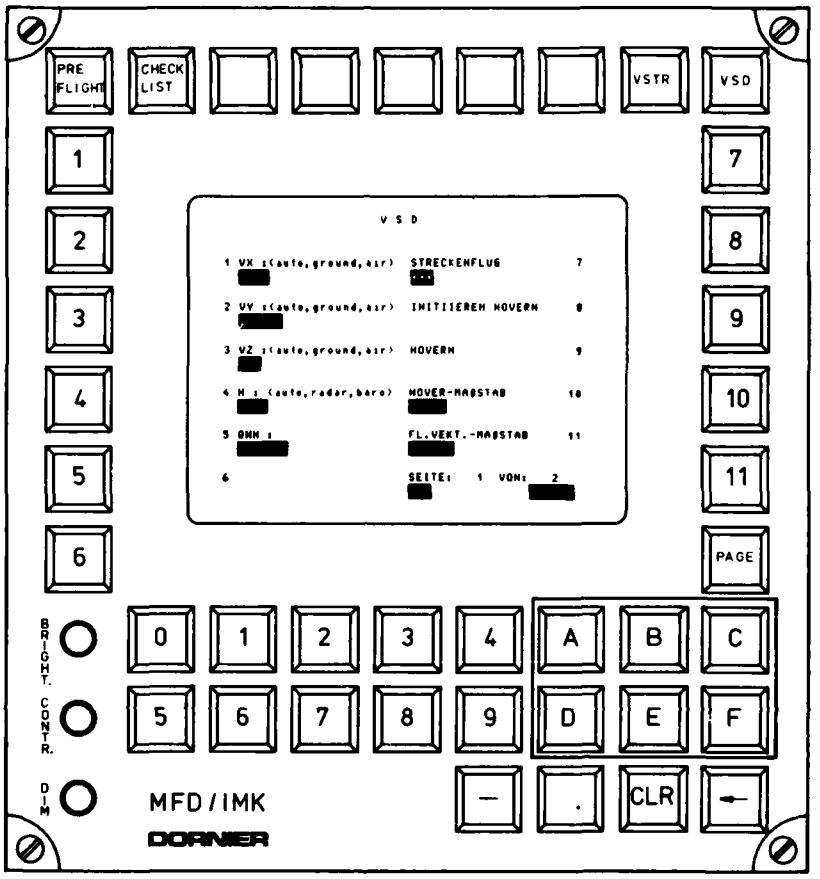


FIG. 2 : MULTI–FUNCTION–DISPLAY (MFD) AND INTEGRATED MULTI–FUNCTION KEY–BOARD (IMK)

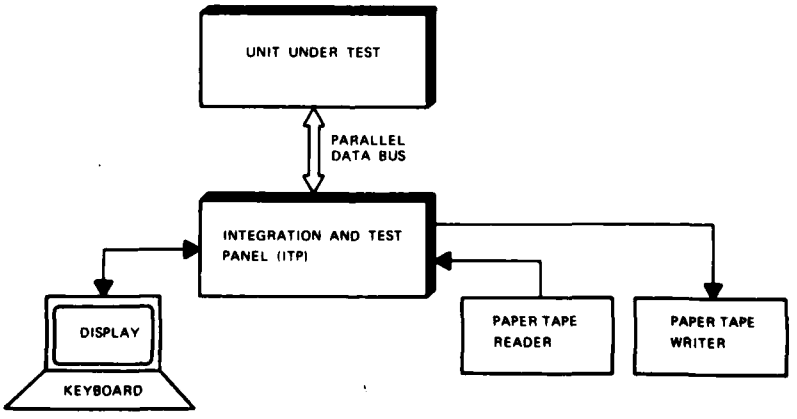


FIG. 3 : TYPICAL ITP CONFIGURATION

SPACE SHUTTLE APPLICATIONS

PART I - Redundant Computer Operation

by
Robert E. Poupard
International Business Machines (IBM)
Federal Systems Division
Owega, New York
United States of America

1. REDUNDANCY REVISITED

IBM's experience in fault-tolerant computers and computer systems has culminated in the definition of the Space Shuttle redundant data processing system and the specification and implementation of the redundancy management techniques for system operation under both normal and failure conditions. IBM's involvement in redundant computers and computer systems dates from the early 1960s and includes processors for NASA's Orbiting Astronomical Observatory (OAO), Saturn, and Skylab; the Air Force Manned Orbiting Laboratory (MOL); and the Army's Tactical Aircraft Guidance System (TAGS) program. Different types of redundancy were employed and applied at different functional levels on these programs.

These programs for which IBM supplied processors have used redundancy at various levels to achieve specific goals. The Saturn computer was internally triply redundant at the module level with voters to achieve a specific reliability goal of 0.995 for 250 hours. The OAO processor was internally quad redundant at the component level to achieve a high probability of a one-year lifetime in orbit (0.9 for one year). The Skylab Apollo Telescope Mount Digital Computer was dual redundant at the box level for high probability of surviving a single failure. Switching was not time critical and the second computer was powered off as a spare in case the operating computer failed. The MOL complex was dual for load sharing primarily, with the capability for each computer to provide a backup for the other. The TAGS program provided a single fault-tolerant triplex digital flight control system for a helicopter.

2. VALUE OF REDUNDANCY

Most everyone agrees that redundancy is a practical way to get increased reliability. The basic problem is to get it to work properly in reaching whatever goals are established: achieving reliability, no single point failures or fault tolerance levels; getting 100 percent coverage (the probability of continuing correct operation given that a failure exists); and, for fault tolerance levels greater than one, removing the failed computer from the system so that a second failure will not take the system down.

Note that all these are independent of the application of the redundant computer/complex. For example, if the application is flight control, the control laws are independent of whether or not they will be executed in a redundant computer complex. Likewise, the redundancy management technique is independent of the control laws. However, some restrictions on redundant system operation may be application dependent. For example, if the application is flight control, then restrictions such as allowable switching transients and the length of time that the control loop may be opened must be established during the dynamic analysis required to define the control laws. These restrictions then are used in defining the redundant configuration and how it will operate.

3. CONFIGURATION

Although IBM pioneered the use of redundancy in computers to increase reliability, the Space Shuttle application of redundancy required new concepts and ideas to meet the NASA requirements of fail operational/fail safe; i.e., the system must provide for safe return of the vehicle and crew after two like failures in the system. The implications of this and derivative requirements on the flight critical computer complex were significant. For example, to meet the requirement to minimize switching transients when a computer failure occurs, it is necessary to have all computers operating and performing the same tasks so that a good computer is "instantaneously" available when another fails. To meet the output time skew requirement between any two computers requires synchronization of the operating computers. To meet the 100 percent FO/FS requirement with no single point failures means that coverage on a failed computer must be 100 percent. This is not possible using BITE and self test alone in today's off-the-shelf computers. Therefore, the computers must operate in a cooperative set and cross check results of critical computations. After all, BITE is only more hardware added to check on the normal functions of a computer. What better, or more complete, BITE is there than another computer to perform the same operations? If the goals is merely to detect a failure (100 percent) then two such units are adequate. However, if the requirement is to continue correct operation with 100 percent surety, after a failure, then three such units are required so that the failed unit can be identified and its results ignored by the system. If it is required that the system continue correct operation after two such failures, then a minimum of four units is required. This reasoning established the four computer redundant set for Shuttle as the prime computational facility. A fifth computer, originally slated for nonflight critical computations, is currently used as an independently programmed backup to the prime set in case of a generic failure of the

prime software. Initially, this was thought to be desirable for Shuttle's approach and landing and orbital flight tests (until confidence was gained in the redundant set concept) since all four prime computers execute exactly the same software. Therefore, a single software error conceivably could cause all prime computers to be in error. NASA and Rockwell later decided that it was desirable to maintain the backup idea through the orbital flight test phase and possibly into the operational phase.

4. FAULT DETECTION AND ISOLATION

Given the four computer redundant set and a two fault tolerance requirement, how is it determined that a computer has failed? If the outputs are discrete signals or pulse type, then the simplest solution is probably to "vote" on the signals. With four computers, the best approach is a two of four voter since it is not necessary then to compute how many computers constitute a majority. The thinking is that if two or more computers agree then they must be right under the assumption of nonsimultaneous failures. Such devices for discrete signals are relatively simple. However, if the output signals are multiple digital words, then the voting process is more complicated and the voters themselves are quite complex. It becomes attractive then to consider using the computers themselves in the voting process. This is done quite simply by passing the desired data for comparison from each computer to all others. In this manner, the powerful logic and computational capabilities of the computer are used to easily determine if its results agree with those from the other computers. This concept is used later in describing the Shuttle technique of fault detection and identification of the failed computer.

To compare computational results, all computers must perform the same computations, in the same sequence, and on the same data. Therefore, a means must be available to synchronize the computers to ensure that the same sensor data is read into all computers at the same time and that they then sequence through the same computations. A straightforward way of synchronizing is to have each computer inform the others when it is ready to perform prespecified tasks. When all are ready, they proceed. Thus synchronization is under software control and is adaptable to meet changing requirements. Other synchronization techniques might involve external interrupts to the computers to start a new cycle and are relatively fixed with respect to a changing environment. Most important is the fact that the software must be synchronized in the computers. It is neither important nor desirable to synchronize the hardware itself. Therefore, a software controlled synchronization technique is desirable.

On the Shuttle system, fortunately, all things are made possible by the requirement that all data communications between system elements be via multiplexed digital data buses. The Shuttle data bus configuration consists of 24 individual buses common to all computers. These buses are functionally distributed for simultaneous or overlapping operation. Functional groups include flight critical (8 buses), intercomputer (5), display (4), mass memory (2), launch (2), payload (2), and instrumentation (1). This configuration offers a ready means of swapping computational results as well as forcing all computers to have the same input data for each cycle. The system configuration in Figure 1 shows the computer complex and the bus interconnections with the remainder of the Shuttle systems. Under software control, each computer can control any or all buses. In practice, each computer controls a prespecified subset of the buses. In this manner, fault tolerant operation of the system is achieved as shown in Figure 2. For the flight critical input channels, a group of four buses, each computer controls one bus and listens on the other three. Control here means simply that only that computer transmits commands on the bus. The transmitter for that bus is disabled in each of the other computers so that they cannot transmit but can receive or listen on the bus. The computer controlling a bus will send a listen or wakeup command over the bus to the other computers before sending the command to the sensor to transmit data. In this manner, all computers in the redundant set receive and store the returned data from the sensor. Since all computers are synchronized, each computer requests data from its sensor simultaneously with the others. Therefore, the three sets of data from the three sensors are time coherent, and no special processing is required to match the three data sets in time.

All critical outputs are voted at the end effector. Normal operation has the effector receiving four inputs and providing one output, although the voted effectors will provide proper outputs with only two inputs. Thus, the two fault tolerant requirement is met on the output side of the computer complex. Each computer in the redundant set normally provides one of the inputs over one of its command channels. There is usually no listen mode for output transactions.

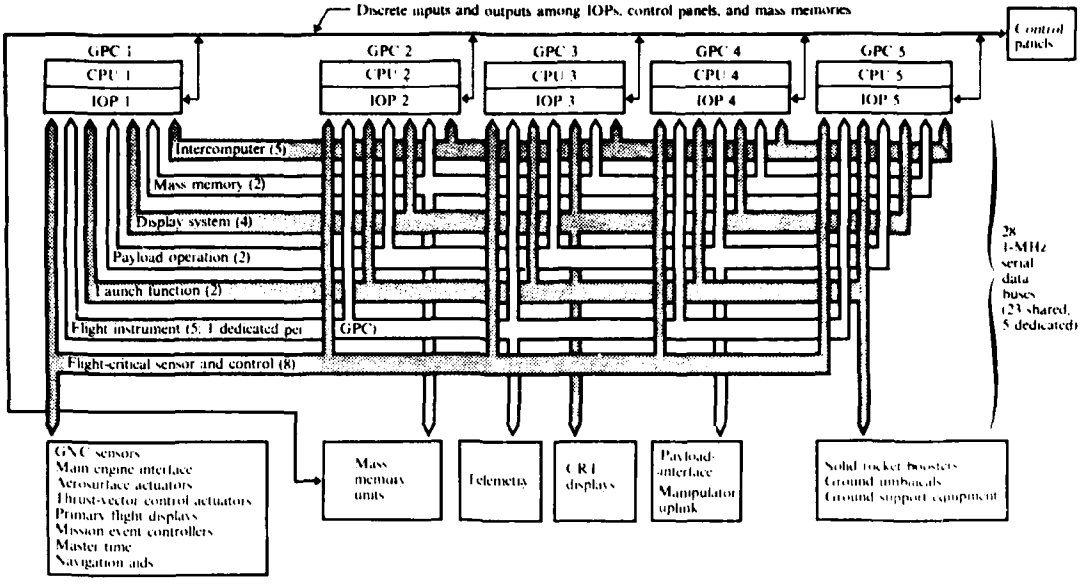


FIGURE 1. SPACE SHUTTLE AVIONICS SYSTEM BLOCK DIAGRAM.

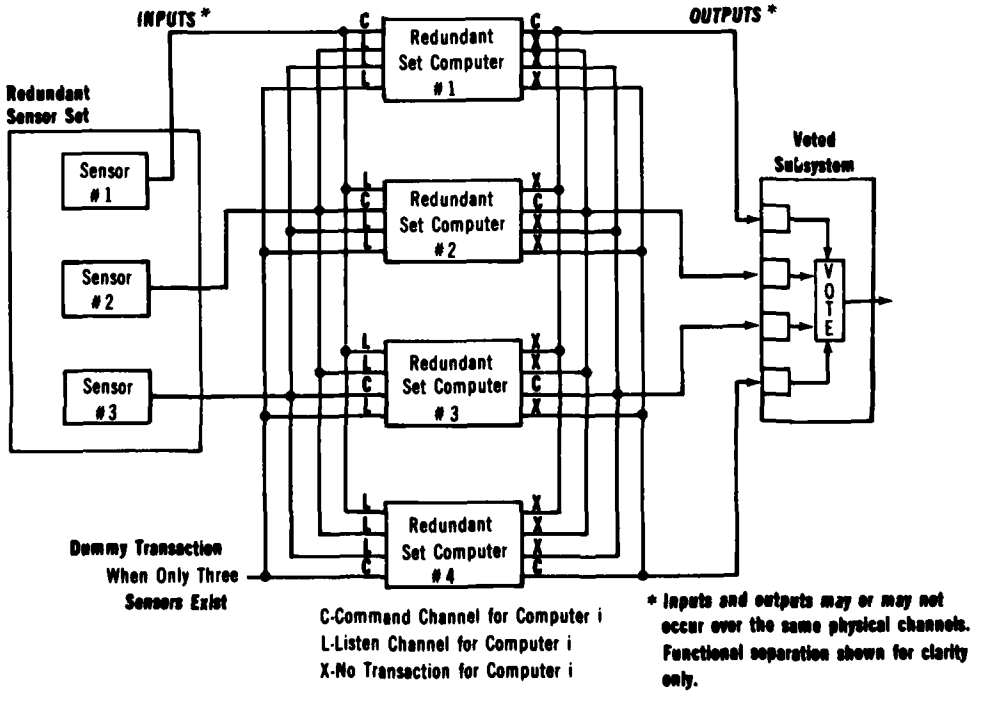


FIGURE 2. REDUNDANT SYSTEM OPERATION

5. SHUTTLE TECHNIQUE

With this background explanation, we have a relatively straightforward means of detecting a computer fault, identifying the faulted computer, and masking the system effects on the faulty computer. Because the downstream voters will filter transiently incorrect outputs from one computer, it is not necessary to instantaneously detect and remove an incorrect output when a computer fails. Thus, the time criticality of comparing computer results among the computers is removed. It is acceptable to wait each cycle until all output commands are computed and transmitted to the subsystems before comparing results. For this comparison, all critical outputs for a cycle are summed and the sum word is compared on the next cycle. Typical quantities making up the sum word are shown in Figure 3. If any computed result is different in one computer then its sum word will not compare with the others. This sum word is transmitted over the critical intercomputer channels once per cycle (25 times per second). The intercomputer channels were selected for this since the proper operation of these channels must be verified as they are critical to normal operation of the redundant set. The proper operation of the remaining flight critical channels is verified by the fact that data is gathered from the sensors and used in computations for the sum word generation. Thus, if the sum word compares, the channels must be operating properly. Other checks such as channel parity, word and bit count, and sync are also used to detect incorrect channel operation.

If a computer fails, the sum word from that computer will differ from those computed by the others. Each computer compares its sum word only with the others. It does not compare each sum word against all others. The failed computer is identified by knowing which intercomputer channel yielded the incorrect word and which computer is in control of that channel. This is important since each computer uses only its knowledge of the system status and configuration to identify the computer(s) with which it does not agree. If a computer disagrees with the sum word from any other computer, it sets a discrete—under software control—to that computer which says, in effect, "I disagree with you." No attempt is made at this point to identify which is incorrect. On the other hand, if a computer receives two or more discretes from its peers, then hardware logic within the computer receiving the fail vote discrets signals that it has failed and, depending on the setting of hardware control latches, may reset its input/output to inhibit further transmission of any outputs. If a computer disagrees with all other computers, then it sets itself failed. It is important to note that the hardware in a computer to indicate the failure is independent of the hardware/software that makes the decision on whether or not that computer agrees with the others. The hardware logic used to indicate a failure is shown in Figure 4. If a computer detects a failure within itself, it will attempt to indicate itself as failed by manipulating the watchdog timer or by letting it time out. If a computer cannot set itself failed, the others will set it failed anyway. Therefore, there is no uncoverage here. The light matrix shown to the right in Figure 4 indicates to the crew the status of each computer's opinion of each of the others. The matrix is five by five since any four of the five actual computers can be selected as the redundant set. The fifth computer is not involved in the comparison process.

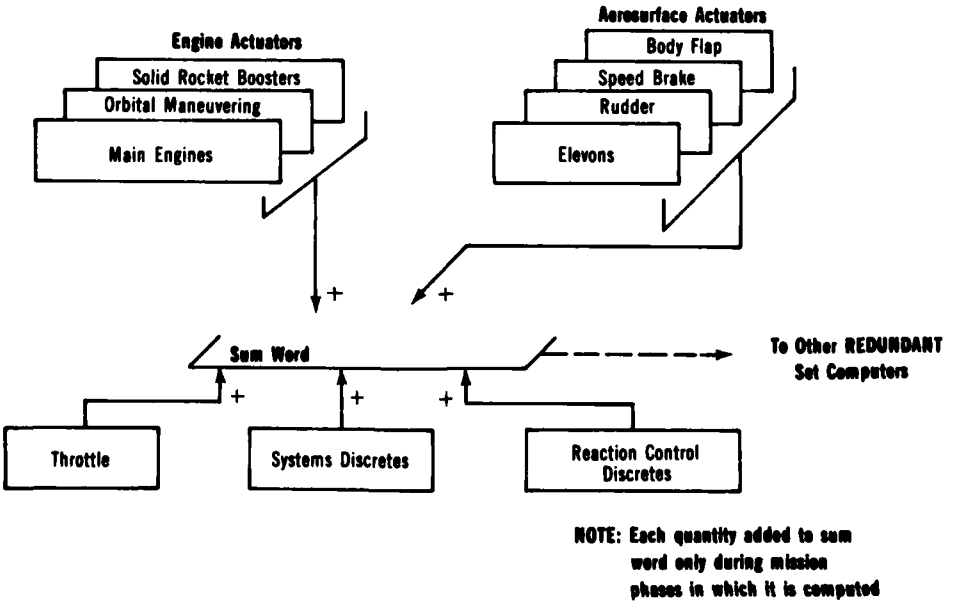


FIGURE 3. TYPICAL SYM WORD MAKEUP

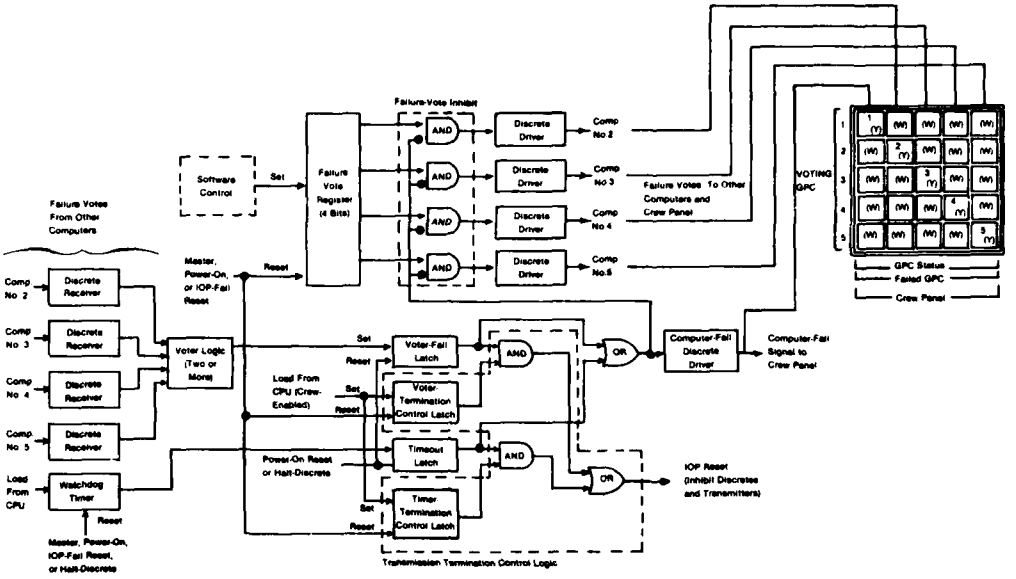


FIGURE 4. DEDICATED REDUNDANCY MANAGEMENT LOGIC, SHOWN FOR COMPUTER 1

6. COVERAGE

A little thought will show that the theoretical coverage for the first and second failure of the four computer redundant set approaches unity. Since the purpose of the redundant set is to provide correct critical outputs in spite of failures, by summing all critical outputs and comparing results all computer failures affecting these outputs should be detected and properly assigned to the failed computer. If any failures occur that do not affect the critical outputs, they are not of immediate concern since they affect only noncritical outputs such as downlink or multifunction displays. The likelihood of having noncritical failures without affecting critical outputs is probably low since the only circuitry not common to both is some memory locations and the input/output channels themselves. Therefore, the conclusion must be that theoretical total coverage is very high and coverage on critical failures approaches one. In practice, the actual coverage depends on both the hardware and software designs. It is possible to design the hardware to eliminate single point failures and uncoverage and then operate the hardware differently than planned by the software design and inject single point failures and/or uncoverage. Careful attention was given to software design for Shuttle to ensure that this did not happen. During the course of software verification for the approach and landing test phase of the Shuttle program, several potential single point failures were discovered and corrected. A task force was convened to review the software design for further single point failures and to recommend corrections for those identified. The result is a total system design with very high coverage and very low probability of single point failures.

7. QUOTE

The demonstration of how far the use of redundant systems in flight critical applications has progressed is best illustrated by Astronaut Gordon Fullerton's remarks after the first approach and landing test flight:

> "...On Free Flight 1, we had a kind of real test of the concept of having the redundant set of computers vote out a failed member and proceed without any glitches—because it did just that right at separation. And, in fact, if we hadn't touched a thing after that, there would have been no difference in the outcome of the flight. We could have gone on and never known it, really. The three remaining computers flew exactly as they should have. So, the concept was proved right there. I personally gained a lot of confidence in that whole idea that I didn't have, never having experienced it before in an aircraft...."

SPACE SHUTTLE APPLICATIONS
Part II - Redundant Computer Software Design and Test

by
Caroline T. Sheridan
International Business Machines (IBM)
Federal Systems Division
Houston, Texas, U.S.A.

1. SUMMARY

When the Enterprise, NASA's Space Shuttle approach and landing test (ALT) vehicle, made its series of successful test flights in 1977 it was assisted by four IBM AP-101 general-purpose computers (GPCs) executing identical real-time control software. In many ways this software was similar to other real-time control systems that use man-machine interface and automatic sensor feedback. It cyclically gathered data from automatic sensors and manual controls; performed navigation, guidance, and flight control operations; and sent control outputs. It drove digital displays for the crew and received inputs through a keyboard. Unlike most other real-time control systems, the Shuttle onboard software was required to produce identical, simultaneous output commands from a redundant set of computers.

In addition, two design characteristics of the system combined with the redundancy requirements produced a unique challenge for software design and development:

A. Asynchronous interrupts. All data that was input to or output by the computer went through its input/output processor (IOP). The IOP processed I/O requests from the central processing unit and interrupted the CPU when each request was complete. The CPU kept processing while its request was being serviced, usually doing something unrelated to the I/O request. So the I/O completion interrupt from the IOP was asynchronous to the CPU processing when it occurred. The other type of asynchronous interrupts that occurred in the CPU were timer interrupts used to initiate cyclic and other time-dependent processing.

B. Multiple asynchronous priority levels. The CPU functions were performed by "processes" that operated at various priority levels. Usually the processes were made ready cyclically at timer interrupts or by the "I/O complete" interrupts; then the highest priority ready process was given control. Interrupt handlers ran at the highest priority, above all processes. The timing of interrupts was not completely random, but there was sufficient variation that any process was subject to being interrupted at any point unless it had disabled interrupts. Since an interrupt might result in higher priority processes being made ready, the interrupted process might not regain control for some time. Long-running, low-priority processes with relatively slow cyclic rates, such as navigation, were interrupted several times during each execution to allow higher priority processes, such as flight control, to execute.

These design characteristics are common to other real-time control systems. Maintaining the integrity of data used at different priority levels is an important consideration in the design of such systems, but not a new problem.

2. SLIVERING

The requirement for identical outputs from the redundant computers, combined with the multiple, asynchronous priority level system introduced a new kind of timing condition, called "slivering." Figure 1 illustrates this condition.

Process B uses the same inputs in both computers, executes the same instructions, and stores the same value in X. The speed of execution, however, and the time of the interrupt invoking process A are not identical in the two computers. The interrupt occurs in GPC 1 after process B has stored X, but in GPC 2 it occurs before process B has reached that instruction. Process A reads X. In GPC 1 it reads the value which has just been stored by process B. In GPC 2 it reads an initial value or one stored in a previous execution of process B. Process A is then operating on different data in the two computers.

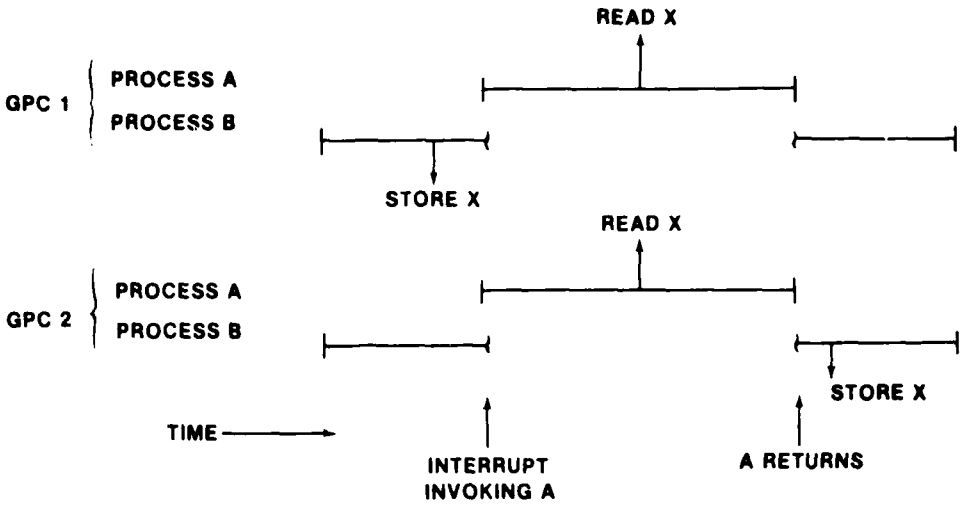


FIGURE 1 SLIVERING

3. PROTECTION OF INTERPROCESS DATA

The protection of "interprocess data" (data passed between priority levels) against slivering involves the use of common techniques for ensuring data integrity in multiple priority level systems, in combination with synchronization between the computers.

Synchronization is software controlled, using synchronization codes transmitted between the computers over discrete lines. At each sync point each computer sends the proper code and reads the sync discrete lines from the other computers to see whether or not they are sending the same code. If they are not, it loops until either all GPCs are sending the proper code or a timeout value is reached. If the timeout is reached, any GPCs that are not sending the proper code have "failed to sync" and are dropped from the redundant set of computers. The timeout value must be large enough to allow for some normal variations in processing between computers but small enough that, should a computer fail, the others would detect the fail-to-sync and continue without too much delay. The value chosen for the Shuttle ALT software was four milliseconds.

With this synchronization technique, a given sync point does not synchronize the computers; it synchronizes the process or interrupt handler that invokes the sync routine. Each priority level must be independently synchronized.

4. SYNC AND DISABLE PROTECTION

The most basic method of protecting interprocess data is "sync and disable" protection. Disabling interrupts at appropriate times is a technique commonly used for maintaining data integrity in single computer systems. In a similar way, with the addition of a sync point, it is used to protect interprocess data against slivering.

The lower priority process involved in the data transfer invokes an SVC routine that synchronizes and then returns control with interrupts disabled. The process then reads or stores the interprocess data before enabling interrupts. The higher priority process, or the highest of several processes that use the same data, does not need to sync and disable.

To understand how this method protects the data, consider the slivering example with the application of sync and disable protection. As shown in Figure 2, process B enters the sync routine in one computer before the interrupt enabling process A. The sync routine allows interrupts up until the time synchronization is achieved. Process A gains control and reads the old value of X in both computers. Then process B regains control, completes the sync and stores a new value in X. If an interrupt were to occur after the sync was successful, then it would not be accepted in any computer until after X had been stored and interrupts enabled.

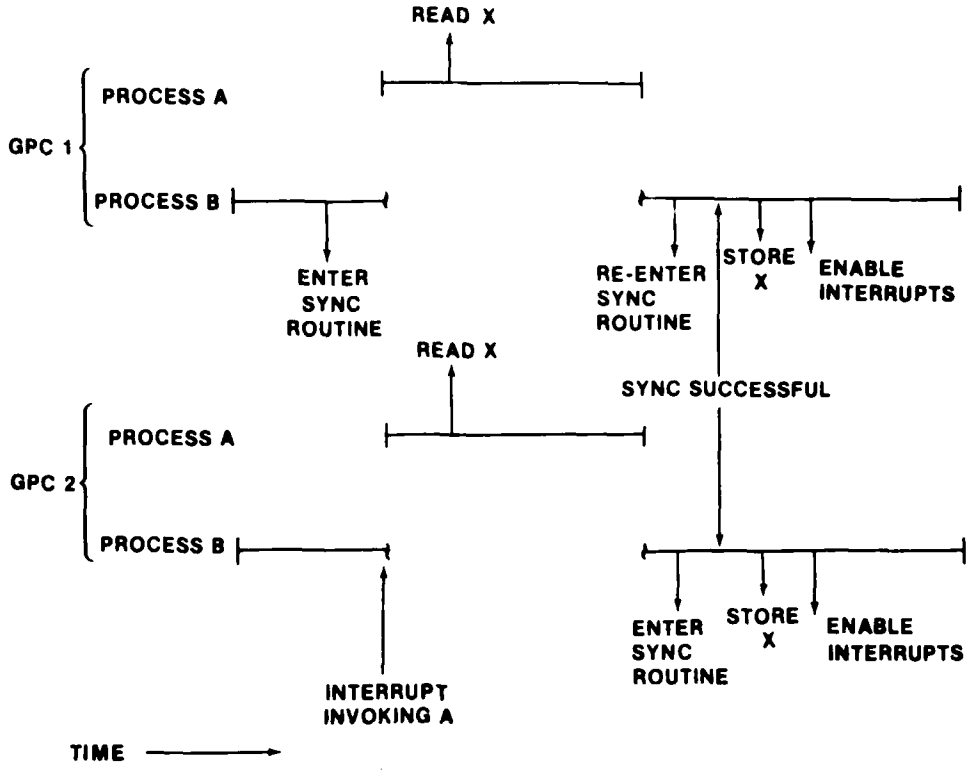


FIGURE 2. SYNC AND DISABLE PROTECTION.

5. OTHER PROTECTION MECHANISMS

Another method of protection is by use of locked data. The data is locked under sync and disable protection. This ensures that the data is locked simultaneously by the same process in all computers. The data can then be accessed with interrupts enabled. Although an interrupt may give control to a higher priority process, that process cannot use the data until it is unlocked.

There is one other general method of protecting interprocess data that has myriad variations. This is by controlling the execution of processes in such a way that the processes using common data cannot be trying to execute at the same time. Since slivering occurs when one process interrupts another that uses the same data, data can be protected by making sure such interruptions do not occur.

6. APPLYING THE TECHNIQUES

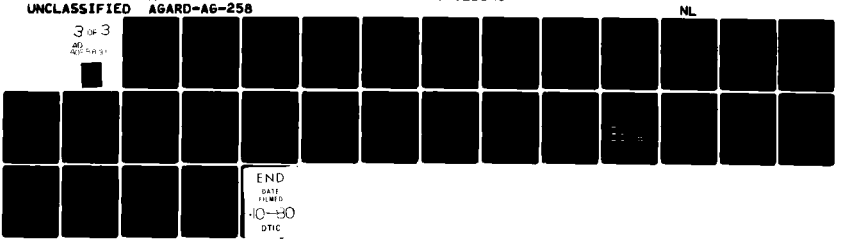
The techniques for protecting interprocess data were simple to use but had to be applied in many places throughout the software. The process of identifying the interprocess parameters, determining whether they needed to be protected, and providing that protection had to be performed at the application level as well as at the operating system level. Wherever interfaces between processess existed, protection of the data had to be considered.

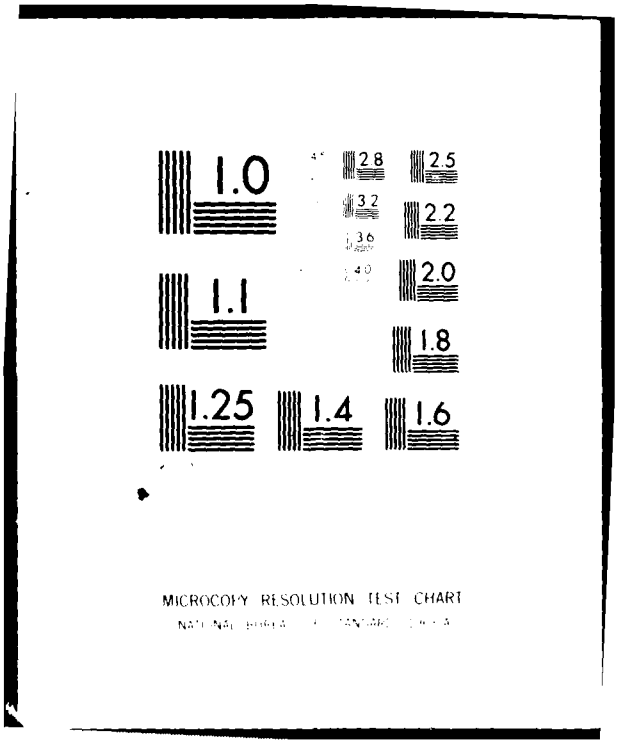
Identifying when protection of data transfers was required was not always easy. Not all the data in the system was required to be identical between computers. Data used in displays for the crew but not entering into the calculation of control outputs was not required to be identical. Processes that only generated displays were not required to maintain identical data. However, they were required to maintain synchronization within the 4-millisecond sync tolerance. Display parameters could be passed between processes without protection in most cases. However, unprotected or nonidentical data must not be used in a test if:

- There were unequal numbers of sync points in the different paths that could be taken as a result of the test, or
- one path was enough longer than another to cause a sync failure.

Because other factors could contribute additional skew between computers, the calculated differences in paths were kept well below the 4-millisecond sync timeout.

AD-A088 631 ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 9/2
GUIDANCE AND CONTROL SOFTWARE,(U)
MAY 80 A O WARD, P F ELZER, H G STUEBING
UNCLASSIFIED AGARD-AG-258 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

7. TESTING

The ALT software was a large system with many interfaces. Moreover, real-time systems, where the timing of interrupts can cause failures in otherwise correct code, are particularly difficult to test. Errors may occur intermittently, infrequently, or never in test runs. When the errors do occur it often takes considerable time to identify the cause.

The ALT onboard software contained potential for two types of timing problems, the usual real-time system problems and the special multicomputer problems. Test runs, even if every path could be run at least once, could not be expected to create all the possible timing situations. A collection of tests, including rigorous code inspection, analysis and test runs of various types, were devised to provide the necessary reliability for a system supporting manned flight.

The initial software testing performed to test the multicomputer system was functional tests of the code that made the multicomputer set run together. This included such things as the I/O processing, the code that synchronized the redundant set initially and the time management system that synchronized the internal computer times to a common time source. Many of these tests were performed during simulated flights that involved the vehicle control applications, so that in addition to testing specific code, they exercised the whole onboard computer software system in a multicomputer environment. These functional tests uncovered some multicomputer problems, including both "hard" failures in the logic and slivering problems with a relatively high probability of occurrence.

A code inspection process was the primary test technique for ensuring the proper protection of interprocess data. All variables that were available to more than one compilation unit were inspected. First the variables used by more than one process were identified. Then the use of each interprocess variable was analyzed to determine whether it needed to be protected and, if so, whether it was properly protected. Systematic procedures, re-inspection, documentation, and reviews of results were used to minimize the possibility of human error allowing problems to go undetected.

The final tests placed severe stresses on the software in various ways. Conditions used to stress pieces of machinery are well known: heat, cold, pressure, vacuum, vibration, etc. Conditions to stress the software in somewhat similar ways were devised. The guidance, navigation, and flight control systems were stressed by simulating flight conditions beyond those expected to be encountered in flight: high winds, errors in sensor hardware at or beyond their allowable limits, and failures of sensors. The redundant processing was stressed by introducing skew between the computers in various ways. These tests performed two important functions. First, they were used to search for problems which might not be revealed by functional tests or code inspections. Second, they demonstrated the reliability of the software and generated confidence in the system.

8. THE OUTCOME

How reliable was the ALT onboard computer software system? Was the extensive testing successful?

At the time the system was delivered it had earned a high degree of confidence from the reviewing community. Hundreds of hours of successful run time had been logged. And the stress tests had shown that the system was very difficult to "break," even deliberately.

The real success criterion of the system must be its performance in actual use. In all of the ALT flights, no software-caused computer failure, failure to sync, or miscomparison of control outputs occurred.

The ALT flights are completed now and the development of the onboard software system for the orbital flight tests (OFT) is underway. The OFT software is larger and more complex than ALT. It will support entire OFT missions from the ground to orbit and back again. The ALT experience has shown that it is possible to make four redundant computers cooperatively control the spacecraft. The hardware and software systems will operate the same way to control OFT and the programming and testing techniques used in ALT will be applied in the development of the new system.

SOFTWARE APPLICATIONS AS DEMONSTRATED IN THE P-3C AVIONICS SYSTEM

by
John W. Heap
Division Superintendent
Combat Systems Software Division
Software and Computer Directorate
U.S. Naval Air Development Center
Warminster, Pennsylvania, U.S.A. 18974

SUMMARY

The U.S. Naval Air Development Center has been involved in the P-3C weapon system development process for the past 18 years. The development of this system has, and continues to be, an evolutionary process in which major updates in system capability are produced every three to five years since fleet introduction of the P-3C weapon system in 1969. One of the major technology roles the Center has played in this system is the development of the core digital data processing avionics subsystem and its attendant system software. This paper will cover the software management methodology developed by the Center for generation of the P-3C system software for the three UPDATE versions of the avionics system. It is not the intention of this paper to discuss in depth the functional capabilities of these versions but rather to discuss the software management process used in accomplishing the development of the ever-expanding system software of this weapon system. Emphasis will be placed on the software development flow process depicting control points and deliverables, standards and objectives set for the software functions and design, the contracting strategy, tools and facilities employed, and lessons learned.

1. BACKGROUND

The enemy submarine threat is, perhaps, the Navy's most urgent and difficult problem. Playing a major role in the Navy's effort to counter this threat is the P-3C Anti-Submarine Warfare (ASW) weapon system. The P-3C is a self-contained system and has been designed to search, detect, localize, classify, track, and destroy enemy submarines. The P-3C aircraft operates from strategically located land bases around the world, monitoring all enemy submarine avenues of egress.

The P-3C is an outgrowth of the U.S. Naval Air Development Center's A-NEW Program. This Program, which began in 1961, successfully established a management and technical team to combine the disciplines of "systems", hardware, software, and human factors engineering. Under Navy Laboratory management and technical control, and supported by an extensive industrial complex, the A-NEW Program developed a series of P-3C laboratory and aircraft testbed feasibility models which culminated with a production system in 1969. The effort successfully married digital technology and user requirements into an integrated, interactive highly effective weapon system. The avionics design that evolved utilized a programmable central digital computer as the "heart" of the system. The central data processing subsystem integrated all the functional subsystems in the avionics system: flight instrumentation, navigation, communications, display and control, acoustic sensors (the prime detection and classification elements of the system), non-acoustic sensors (elements such as radar and electronic surveillance), and armament and ordnance. Additionally, it provided an effective aircrew and avionics interplay management scheme. The computer performed the majority of the data management and integration functions by processing subsystem data for periodic and demand computations, control and display, storage, and eventual retrieval. This system design approach provided aircrews with five times more time for critical decision making – as opposed to previous airborne ASW weapon systems that consumed virtually all the aircrew's time in manual data gathering, status checking, and computation chores.

As new technology develops for submarine systems, airborne ASW system developments must have a clearly defined program for future improvements. For the P-3C weapon system the P-3C UPDATE Program provides a continuing series of functional enhancements to counter the continued improvement in enemy submarine capabilities. In the interest of maintaining a uniform avionics design philosophy, continuing P-3C UPDATE analysis, system development, and system software development and fleet support has been the responsibility of the U.S. Naval Air Development Center. There are presently three UPDATE versions of the P-3C weapon system. Briefly described they are:

- UPDATE I – an expansion of the data processing subsystem in word storage capability (memory) and the number of input and output channels, a complete redesign and rewrite of the Mission Software, and the addition of OMEGA navigation.
- UPDATE II – continued sensor improvements to include a long needed sonobuoy location system.
- UPDATE III – a completely new acoustic processing subsystem, new digital magnetic tape recorder equipment, and additional sensor improvements.

The past 20 years has seen the U.S. Naval Air Development Center become deeply involved in the "systems" development process of Navy airborne weapon systems. During the 1960's only a few of these systems were utilizing large digital processing avionic equipments with their attendant real-time system software packages. The P-3C was an early example of a weapon system utilizing extensive digital processing and Mission (operational) Software. This early large-scale software application was leading the trend towards large central and/or distributed processing systems as the best technical approach to future weapon system integration, automation and logical expansion. Through the 1970's the trend has evolved to the point where the majority of P-3C mission scenarios are dependent on the real-time Mission Software which is embedded in the avionics system. System Test software for determining avionics system readiness and for use in avionics diagnostic maintenance, and program generation center and integration facility support software have all rapidly expanded in parallel with the growth in Mission Software.

Because of the prominent role software now plays in the P-3C weapon system and the historical software management problems exhibited in the past, it was critical that Center management establish a software development methodology which would better assure successful weapon system development, introduction, operation and readiness throughout the weapon system life cycle. This paper will describe the process which has so far proved highly successful since 1972 in managing the development and fleet maintenance of the P-3C UPDATE system software. The methodology presented is oriented to Mission Software development as opposed to System Test software, although certain general issues and approaches apply equally to both types of software. As a point of reference the reader must recognize that when this methodology was first proposed in the 1972 time frame it was prior to extensive publication of technology efforts on software engineering for real-time weapon systems. Therefore, many items which were innovative during that era are commonplace today.

2. P-3C WEAPON SYSTEM DESCRIPTION

The enemy submarine's capability to operate deeper and quieter at greater speeds is continuously improving. At the same time, the technology required to cope with this increasing threat is becoming more complex. With the increasing threat and more complex systems it was essential that the Navy plan incremental updates in capability of its P-3C weapon system to cope with the problem. It must be recognized that the P-3C airframe has more longevity than its attendant avionics system. A thorough discussion of the functional and system design features of the P-3C already fills books and is not the intention of this paper. However, it would be of general assistance to present a brief overview of the P-3C, and to include in some detail the data processing subsystem for a better understanding and appreciation of the past, present, and future task of managing the development and fleet support of the system software.

The P-3C is a land-based maritime patrol aircraft with a primary mission to search, detect, localize, classify, track and destroy enemy submarines. It can also search, detect, localize, classify and track friendly vessels. The weapon system consists of a four engine turboprop with computer integrated avionics consisting of flight control, navigation, communications, aircrew display and control, sensors, tactics, and armament and ordnance subsystems. To fulfill the mission requirements, the aircraft exhibits the following operational characteristics. These figures are only approximations for use as a reference point. The maximum gross weight of the aircraft is 135,000 pounds, with a zero fuel weight of 75,000 pounds. The maximum range capability with four engines is 4,000 nautical miles. The maximum indicated airspeed at sea level is 400 knots. A normal ASW mission profile would be takeoff – high altitude enroute (30,000 feet) – low altitude mission conduct (1,000 feet) – high altitude return (30,000 feet) – landing, with a time on station (mission conduct) of five hours.

From the standpoint of historical sensor and sub-unit development and for the purpose of good organization, the P-3C avionics system has been categorized into functional subsystems with the subsystems consisting of both hardware and software elements. Together, the following subsystems comprise the total "avionics system" of the P-3C weapon system:

Flight Instrumentation
Navigation
Communication
Data Processing
Display and Control
Acoustic
Non-Acoustic
Armament/Ordnance

The aircrew functions as the managers of the weapon system. Their task encompasses mission planning, crew station operational execution, evaluation of data, decision-making and command. The man-machine interface has been designed to make maximum use of the aircrew's "thinking" abilities (actions requiring quality knowledge) and the machine's data processing and handling abilities (actions requiring quantity knowledge). The aircrew has a complement of nine: pilot, co-pilot, flight engineer, tactical coordinator, navigation/communication operator, sensor station 1 (acoustic) operator, sensor station 2 (acoustic) operator, sensor station 3 (non-acoustic) operator, and armament/ordnance operator. Figure 1 presents a pictorial of the P-3C weapon system and, hopefully, serves to give definition and perspective to the words weapon system, avionics system, system and subsystem.

The Navy's solution of the ASW technical problem is to develop capabilities which continue to improve upon the weapon system's ability to determine an enemy submarine's position, course, and speed. The success of the subject mission is therefore dependent on: the ability of the weapon system to position the aircraft and sensors; the capability of the sensors to detect the target; the ability of the weapon system to collect and analyze the data, and make decisions based on conclusions reached; and, the capability of the weapon system to react to any commands and be flexible to varying mission scenarios.

As is shown, the data processing subsystem is the "heart" of the P-3C. Its function is to provide efficient avionic system operation through integration of the various subsystems. It must provide the necessary commands, computational ability and data storage to enable the total weapon system to perform its many diverse functions. The data processing subsystem consists of the computer(s), peripheral equipment, and computer software programs. Figure 2 presents an overview of the subsystem and also depicts the growth areas introduced in the UPDATE versions by labeling the new equipments. Figure 3, in addition to further explaining the data processing computer and peripheral equipment changes, also indicates the various function and software program capabilities of the basic P-3C and the three UPDATE versions. The following paragraphs will briefly elaborate on the major elements within the data processing subsystem.

The CP-901 is a miniaturized, general-purpose, stored-memory computer. It is manufactured by the UNIVAC Division of Sperry Rand Corporation (with a military designation of AN/ASQ-114). The computer has a core memory capacity for 64,000 30-bit words, and has a 2 microsecond cycle time which can be reduced through utilization of an overlap feature.

The peripheral equipment consist of: auxiliary storage devices (magnetic drum and tape), for loading computer memory with data and instructions; a data multiplexer, to provide the computer with additional input and output channel capacity; and, a high speed printer, for a hard copy of computer data. The data processing subsystem logic units 1, 2, and 3 can really be considered as interface devices and placed functionally in a supporting role with the other subsystems such as navigation and acoustics. However, for simplicity of understanding they are included in figure 2 as part of the data processing subsystem to show how this subsystem interfaces with the total avionics system.

The Mission Software is a set of computer programs which integrates the man and the avionics system's hardware, and enables the resultant man-machine system to fulfill its mission requirements. It is sometimes referred to as operational software. The software is executed during all phases of the mission. It is divided into application modules which have parallel functions to the aforementioned subsystems: navigation, communication, flight instrumentation, display and control, acoustic, non-acoustic and armament/ordnance. Additionally, a tactics application module is the integrating function of all of the above subsystems. Implicitly included within the computer software program are the executive control module required to manage the data processing system, the initialization loader module required for loading the software, the recovery module required for software recovery in case of the need for a computer restart, and the data retrieval module required for postflight reduction and analysis.

The System Test software provides a preflight systems readiness check (system go or no-go) and maintenance diagnostic mechanism to verify the operation of all equipments under the control of or monitored by the data processing subsystem. The software is used extensively as an aircraft maintenance tool for the purpose of detecting and localizing hardware failures. As a fault detection tool, it exercises the man-machine interfaces such as display presentations, switch operations, and indicator functions. Internal hardware test loops and computer controlled built-in tests may also be exercised as allowed by the individual equipments or subsystems.

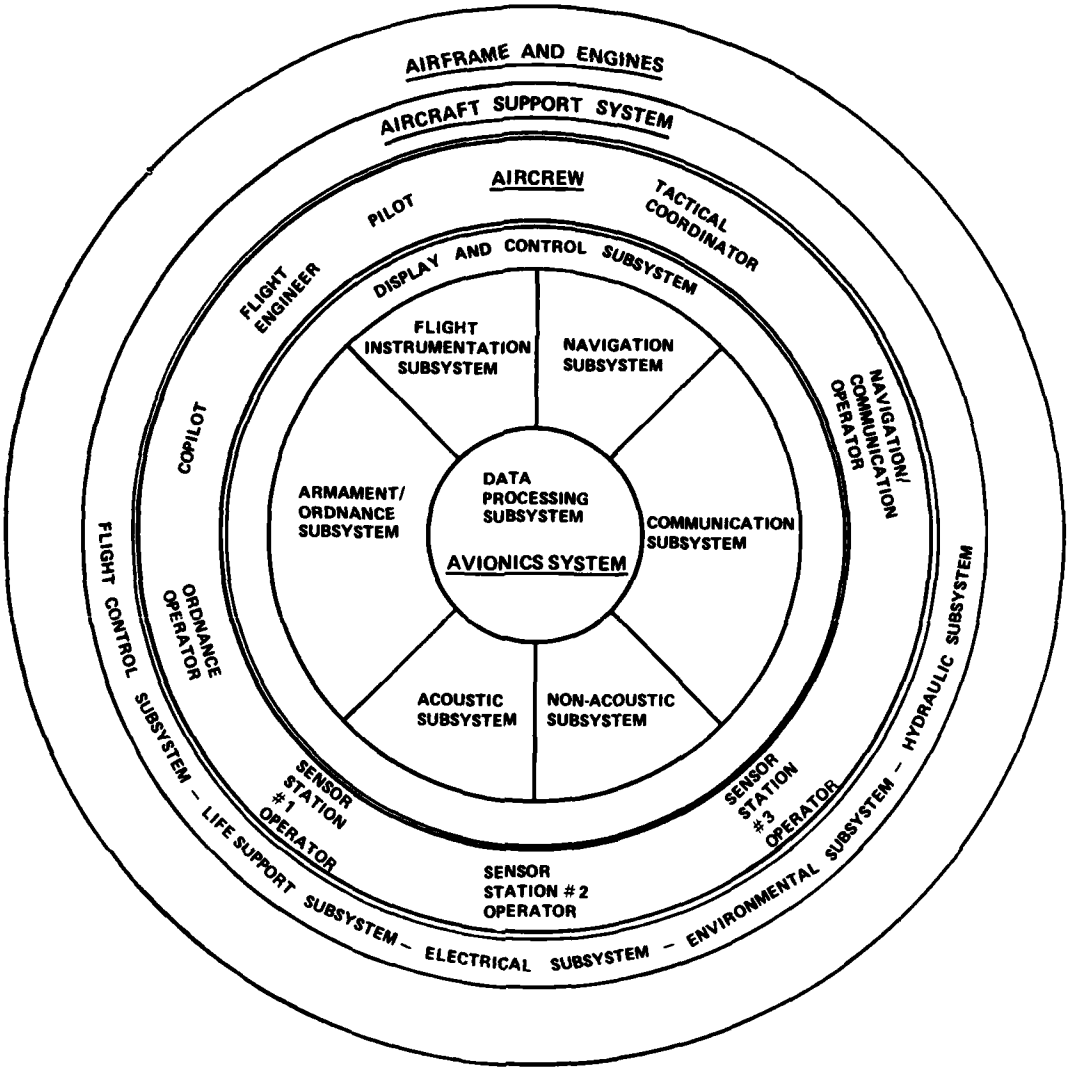


FIGURE 1 – P-3C Weapon System

The term support software refers to all remaining software programs used to assist in the development and fleet maintenance of the Mission and System Test software and includes avionic system resident analysis aids, facility resident generation, equipment acceptance, system/subsystem integration, configuration management, stimulation, and analysis related software programs. Support software used in UPDATE Programs will be described in the Software Development Process section of this paper.

The critical path of any P-3C UPDATE avionics improvement program hinges on the Center's ability to manage changes to the Mission Software. The remainder of this paper will discuss the software development methodology applied first to the UPDATE I Program and continued for subsequent UPDATE Programs with some noted changes.

3. SOFTWARE MANAGEMENT ISSUES

Historically, a few moments should be taken to reflect upon the P-3C system software development process prior to the UPDATE era. There were many managerial and technical problems, the greatest being a lack of a well-defined structured and disciplined software development approach. The process of building the software products was loosely structured and not integrated in any tightly controlled fashion. Software management of large scale P-3C systems was in its infancy. The testbed aircraft was the only software debug, system integration, and test facility. There were no other software facilities or tools, other than the rather slow and cumbersome tape oriented compiler and tape generation facility. Weapon system and subsystem requirements (that is, documentation defining subsystems such as communication and navigation) did not exist. Mission Software functional requirements (documents which describe tasks and/or algorithms to be performed by the software and also adjudicate functions between the man and the machine) were not precise and lacked some of the diagrams and detailed explanations of events/actions/equations required. The testing and product verification process likewise had no formally structured sequence for checking the quality of the software product – whether at the system functional level, system design level, software functional level, or software design level. Minimal laboratory support facilities resulted in all the testing being performed right on the testbed aircraft. The contracting strategy also did not lend itself to strong management control. A single contractor wrote the functional requirements documents, specified the software design, implemented the code, and tested the product for conformance with the requirements. In reality, the Center was not effectively controlling the development process or, in fact, did not fully technically "own" the product since some of the knowledge of the system software and its attendant requirements was possessed only by the contractor. The prime software contractor role is not a bad concept in and of itself but the Center's software products, under this concept, must be controlled in some positive manner in the areas of quality, schedule and cost which, in the case

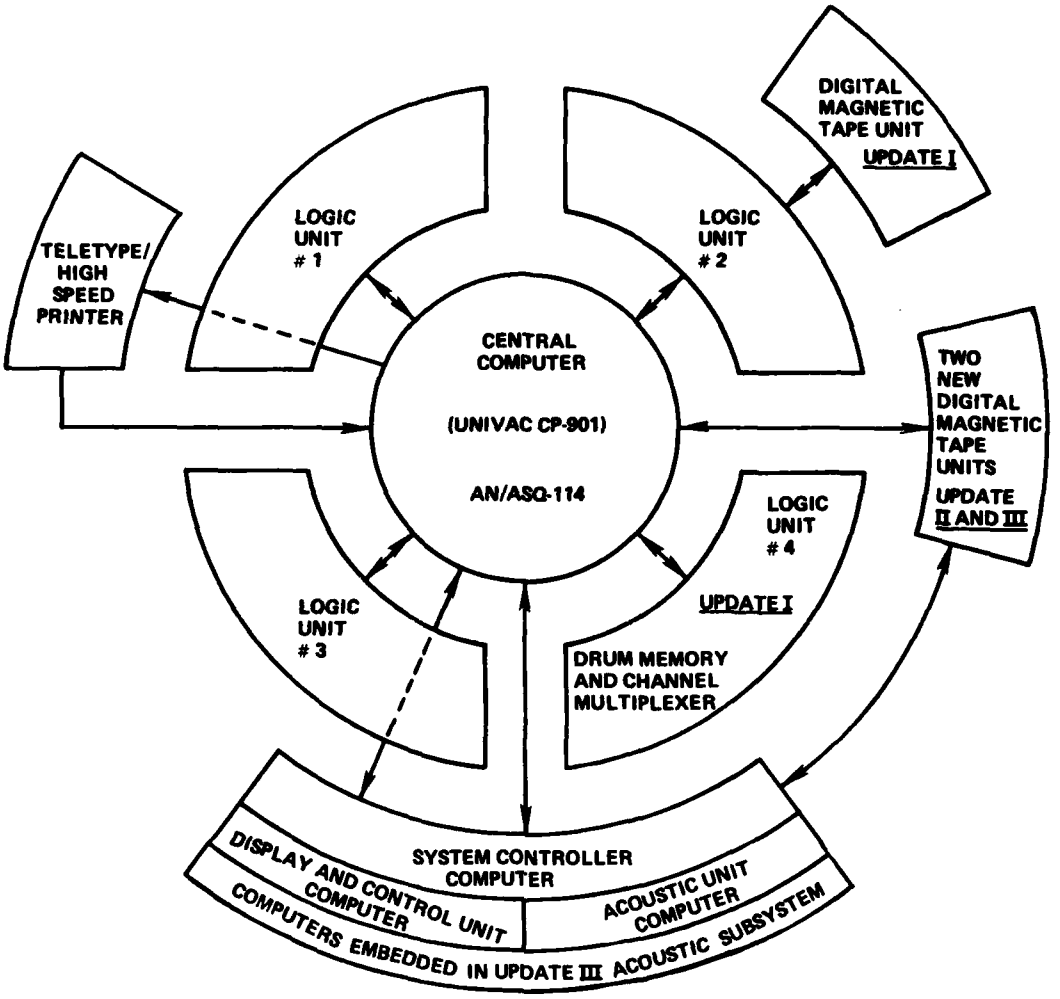


FIGURE 2 – P-3C Data Processing Subsystem(s)

of P-3C software development history prior to the UPDATE era were only under fair control in these areas. As can be readily seen from these historical difficulties, system software management required a great deal of maturing and thought before the P-3C weapon system could proceed into any type of major enhancement program.

Good management dictates that before any approach to problem solving is established, you must first derive a set of program objectives and contemplate pitfalls, using all historical information. Through the process of trial and error, the Center has been able to define a set of overall systems software development objectives. These objectives logically flowed from previous A-NEW Program and P-3C Program experience. The system software development objectives aim to provide the Center with:

- The ability to state its functional requirements and design needs in meaningful terms to industry.
- The ability to measure industry proposals and efforts against in-house knowledge and development testbed models.
- The ability to make knowledgeable decisions in software procurements.
- The ability to accept, in-house, technical responsibility for any element of the work effort as required.
- The establishment of a known system software development methodology which can be applied to many programs and be readily understood by both Navy and industry.

Even though these objectives are biased towards a Navy laboratory performing in a prime contractor role, the same objectives can easily be used by any agency or company that is building or buying software products. To carry out these objectives the Center must have a strong in-house technical and managerial team. An agency could not attempt to utilize the approach put forth by this paper without having a team of employees which could address the numerous technical issues and perform the corresponding technical managing of the software products. The Center has had to develop a whole new series of professional employees called Software Engineers to deal with these technical and managerial tasks.

Although the term Software Engineer is not new it would be helpful to discuss what the Center is doing in this professional category and how this profession applies to the development of weapon system software. Software Engineering is the emerging technology

P-3C VERSIONS

	BASIC (1969)	UPDATE I (1975)	UPDATE II (1978)	UPDATE III (1983-PLANNED)
SYSTEM FUNCTIONAL CAPABILITY	BASIC SET OF: DISPLAY & CONTROL TACTICS NAVIGATION COMMUNICATIONS ACOUSTICS FLIGHT INSTRU. NON-ACOUSTICS ARMAMENT/ ORDNANCE	ADD: OMEGA NAVIGATION MODEL IV COMM. LINK IMPROVED: STEERING TACTICS ACOUSTICS	ADD: SONOBUOY REFERENCE SYSTEM INFRARED DETECTION SYSTEM IMPROVED: ACOUSTICS	ADD: TOTALLY NEW ADVANCED ACOUSTICS SUBSYSTEM
DATA PROCESSING SUBSYSTEM				
COMPUTER(S) PLUS MAIN FEATURE(S)	CP-901 TAPE OVERLAY CAPABILITY IN RECENT YEARS	CP-901 DYNAMIC ACCESS DRUM MEMORY AND EXPANDED INPUT AND OUTPUT CHANNEL CAPABILITY	CP-901 CONTINUE UPDATE I FEATURES, PLUS NEW DIGITAL MAGNETIC TAPE UNITS (BACK FIT)	CP-901 CONTINUE UPDATE I AND II FEATURES, MAG. TAPES (FORWARD FIT), PLUS 3 EMBEDDED COMPUTERS IN ACOUSTICS SUBSYSTEM
MEMORY RESOURCE	65K COMPUTER CORE	CONTINUE BASIC, PLUS 363K DRUM	CONTINUE UPDATE I	CONTINUE UPDATE I PLUS 224K COMBINED CAPABILITY OF 3 EMBEDDED COMPUTERS (EST.)
SOFTWARE PROGRAM SIZE				
MISSION	72K	273K	310K	500K (EST.)
SYSTEM TEST	450K	600K	600K	1,700K (EST.)

FIGURE 3 – Data Processing Subsystem Functional Capability and Processing Attributes

and corresponding professional career for performing the technical tasking associated with development and fleet support of software products. This new professional field encompasses taking weapon system mission and readiness requirements and, by marrying computer science technology, human behavioral science, and digital electronic engineering knowledge, develop real-time system software products within the cost, schedule and performance envelopes specified by program managers. The tasks to be performed in building these software products fit into the classical definition of tasks to be performed by an engineer, as opposed to a scientist, whereas the professional is given requirements for a product and must supply state-of-the-art software knowledge within a well-defined cost, schedule, and marketable performance envelope. The Center has created, in response to the pressing need, a career series titled "Software Engineering". Entrance level is by possessing a bachelor's degree in Electronic Engineering, Mathematics, Physics, or Computer Science. Professional positions have been established from junior (appentice) Software Engineer through supervisory Software Engineer. The creation of this series has been a major step forward in developing an appropriate professional environment for building strong in-house teams to deal with the subject software issues, approach and process. It should be noted that this career series is not yet formally recognized by government job classification authorities but they have informally encouraged the use of this professional title since it most accurately describes the jobs to be performed.

Rather than proceed into detail on the objectives, it would be better to present a list of historical software development pitfalls. After presenting this list it will be easy to determine the reasons for a specific objective. The following list is quite consolidated but will serve to emphasize the large variety of problems contributed to managing software development efforts:

1. Lack of definitized requirements – Many of the P-3C requirements were only verbal statements from knowledgeable members of the ASW community. Although they understood what they wanted the weapon system to do, those people responsible for building the product greatly lacked the fleet operational experience base of reference, so the requirements statements were perceived quite differently by system and software builders. The operational community conversely lacked the technical knowledge of what is possible to accomplish or build, and in what time frames. The UPDATE I Program therefore created a comprehensive set of requirements documents, which demanded program front end documentation discipline, for both adjudication of system requirements (definitizing subsystems – hardware, software, crewmen) and software requirements (definitizing software modules and human responses).

2. Undisciplined software development process – The process must be well defined and the development team be familiar with its flow, control points, critical paths and deliverables. The process must be understood by management and worker alike, across both Navy and industry. The UPDATE software development methodology process was, for many years, shown on a wall pictorial in the UPDATE Program Control Room. Repeated briefings by senior project personnel to all team members was considered one of the unifying forces which started the UPDATE Navy and industry team towards common objectives. A representative pictorial of this process is presented in a following section of this paper. It cannot be stressed enough the importance of the whole team having a common understanding of the process.

3. Deference of system problems to software – It is common knowledge today that many of the large software development program problems in the past were really system engineering problems which went unsolved and created havoc during software integration.

The software development process is not that flexible that it can absorb such difficulties and still control cost, schedule and performance. The UPDATE Program attempted to reduce this cascading problem effect by forcing as much as possible solutions to systems problems at the front end of the avionics system design phase. System level documentation set forth what was expected from each of the avionic subsystems. Interface design documents and integration software was generated to control the input and output signals/functions to the data processing subsystem. Finally, all software requirements were put in numerous volumes (UPDATE I had 20 volumes) to bound those elements of the avionics system which required software for implementation.

4. Inadequate support systems – There needs to be a good set of facilities and support tools for building software. These support elements should have three strong characteristics. First, the set of software facilities and tools should be "comprehensive" and a large percentage of the avionics system development costs should be devoted to supplying these support elements. Generally, the stronger the set of support elements the lower the risk in software development. Secondly, the support elements must be "integrated", that is, fit into the logical flow process in some building block manner. The integrated effect should provide, as an objective, to present managers and technical staff alike visibility in the various stages of software design, code, generate, integrate, and test. Another way to state the case for visibility is that in combination the software support elements must present checks and balances over the development process – exposing any problems in the areas of system function, system design, software function, and software design. Finally, the support elements must be "reliable". There must be a considerable investment of resources to properly validate each facility and tool. If the in-house technical team as well as other agencies and industry, do not have a high degree of confidence in the support elements, management's efforts will be drained away in solving arguments between support facility personnel and the user/software building community. But more importantly is the fact that the Center's in-house technical staff must have good performance measurement tools to insure a quality product within cost and schedule.

5. No anticipation of problems and inability to manage change – For some strange reason, despite experience, all people planning large software developments present a program of minimal time and cost with maximum performance, and act as if they will be spared from any problems and that the program and functional requirements will never change. Of course all of the above horrors do happen and the total software development process must be so structured as to allow breakpoints for re-entering with new requirements and program changes. It is very important that the process also set up points for a functional freeze and design freeze, which once passed will not permit any major changes without a total replanning of the development of the software product.

6. Poor communications – When building system software, there is a tremendous mix of technical disciplines that must be able to communicate knowledge in an effective manner. The effectiveness of this communication is generally the largest but least obvious problem managers have in building software. There is a difficult "translation" problem when one considers the total path an idea (requirement) must travel before it is working as just one small element of a large and complex system. The user ASW community must state their requirements clearly – the systems engineers must know enough about ASW to understand and put the requirements into technical terms and assess the technical feasibility of implementing or not implementing the requirement; the Software Engineer must possess similar ASW knowledge as the systems engineer but must perform the assessment and implementation tradeoff within the data processing domain; the programmer need not know all the engineering tradeoffs that have been performed but must have the design linkages and requirements clearly stated so as to proceed to detailed software design and coding. This explanation emphasizes the interdiscipline communications which must exist in building large software systems like the P-3C UPDATE versions. Also paramount for good software development communications is the need for all the teams to have a common understanding of the process and terms used in the process. Such items as Preliminary Design Review (PDR) and Critical Design Review (CDR) must be understood in terms of their content and meaning to the overall process.

7. Weak Procurement Strategies – A section of this paper has been dedicated to contracting strategy. However, three main historical problem areas will be stated here since they form the foundation of the UPDATE procurement strategy. One is that without "strong in-house software knowledge", it would be extremely difficult to control the software building or buying process. The remaining two involve the contractor hired to build the software. Many often plot a business strategy to "buy in" on the product, that is, deliberately low bid the job in order to gain inside knowledge and control over the process and product. Secondly, contractors on major software products which feel no threat of competition tend to become complacent which sometimes reduces the quality of their product and responsiveness to Center control.

8. Decentralized development and support – The management approach taken by this Center is that by having one agency perform both software development and fleet support, tremendous savings in money, high quality products, and fleet responsiveness can be gained for both jobs. The two factors forming the basis of this conclusion are the already developed and proficient in-house professional staff with corporate knowledge of the P-3C UPDATE software, and the already developed and validated support facilities, thereby eliminating very expensive duplication of these items.

It is hoped that this discussion of the software development objectives and pitfalls will help explain why the P-3C UPDATE Program adapted many of the technical and management strategies presented in the following sections. The explanations offered with each of the pitfalls implies many of the actions management took or conditions guarded against while building software.

4. SOFTWARE MANAGEMENT APPROACH

Although the concept of having the P-3C capabilities enhanced every several years is seen as second nature today, this phased UPDATE delivery approach had to be developed and accepted. When the P-3C was introducted into the fleet in 1969, or when it was clear in 1970/1971 that there were many sensor and system integration improvements which had proven feasibility and warranted inclusion in the avionics, there was a prevailing management philosophy to try for one large upgrade in the future which encompassed all the improvements. The Center felt that it was much more technically and managerially sound to produce phased deliveries of capabilities, one phase building upon the foundation of the preceding phase. The first phase, being UPDATE I, would establish both a data processing subsystem hardware and software suite which would have reserve capacity and lend itself to a building block concept. The succeeding phases would concentrate on sensor improvements in areas needed to keep pace with the enemy submarine threat. The Center's strategy was accepted and today we continue to see deliveries of the UPDATE versions to the fleet. In conjunction with the evolutionary building block concept was the Center's desire to perform the lead system and software development role for the avionics system, a task normally performed by a prime contractor. Early in 1972, Navy management gave the Center the go ahead to proceed in this unique role for a Navy laboratory. The following paragraphs depict some of the software management approaches used in performing this task.

As discussed earlier one of the key points to building or buying software is to develop a strong in-house software team which possesses good corporate knowledge of the P-3C data processing subsystem. In building this team the first step was in establishing

professional requirements for positions at all levels in the Software Engineering field. While bringing this task to an acceptable degree of maturity the Center likewise worked on developing corporate knowledge specifically relating to the P-3C. The following constitutes a list of knowledge required of an in-house full performance level (fully trained professional) Software Engineer working on a P-3C UPDATE Program.

1. An understanding of the ASW mission requirements and its corresponding real world environment.

2. A working knowledge of the P-3C data processing subsystem and avionic system architecture. This encompasses knowing the physical hardware suite and data processing input/output layout, the instruction set of the CP-901 computer (the embedded computers in the UPDATE III acoustic subsystem), the assembly language(s) of the computer(s) and the Navy high order language used in the CP-901 (CMS-2Q), and the features of the executive control program(s) and data base design(s).

3. The ability to know a functional flow through the entire avionics system. An example of knowing a functional flow would be by being able to trace a function such as aircraft true heading – possess the knowledge of how it laces throughout the system, both in hardware and software, and what interfaces, switches, displays and equations are affecting its use and accuracy.

4. The final knowledge category differs for each individual depending both on prior professional background and management specified subsystem needs for the particular UPDATE version. The requirement is to understand a total avionics subsystem function such as navigation, communications, tactics, and electronic surveillance. The Software Engineers are then in a position to develop with confidence the corresponding software portion of an avionics subsystem. They resolve functional ambiguities and interface problems, and implement very specific and often complex algorithms with their particular subsystem knowledge. Management concentrates this sort of professional talent towards the new data processing or sensor capabilities being added to one of the UPDATE versions.

Management has to develop a clear picture of the contract strategy used in protecting the Navy's long-term interests. The basis of the strategy is encompassed in the following procurement plan. Major tasking phases of the software development process are broken out into well defined responsibilities, and contracts let to multiple contractors in such a manner as to give management good visibility of each participant's work. It is also important that each contractor's product have high visibility in the overall development process so corporate responsibility and pride can be clearly displayed. Finally, through strong in-house knowledge and good support contractors the Center creates a competitive atmosphere for the prime software implementation contractor. This procurement plan is revisited every so often to make sure the existing set of contractors are cost effectively working in accordance with the strategy.

One of the major control features the Center employed in the lead role as prime software developer was to build all the software support facilities and tools so that they would be physically located on Center and under in-house control. Equally important to this approach is to have all the support items developed and managed by a separate team of in-house and support contractors other than the teams building the Mission and System Test software. This is done to further insure that management has a good set of checks and balances over the support items, and that there is minimal technical bias in their design and operation. There are several attractive management features of having the support items in-house. The support contractors and prime software implementation contractor all have to request and use the facilities under Center control. As their management and technical staff interact with government provided support, the Center obtains valuable information on the technical progress the product is making as it advances through the software building block process. The contractors lose to some degree their ability to hide problems, whether they be technical, cost or schedule. A strong feature of the Navy "owning" the support facilities is that it prevents any one contractor from capturing the support items and thereby not limiting competition.

Paramount to performing any major software development effort is the need to understand and be in compliance with Department of Defense and Department of Navy software standards and guidelines. The methodology outlined in this paper is fully in conformance with these publications and the Center as a Navy agency objectively carries out the policies put forth without having any conflict of interest, which a contractor might have because of special purpose company interests. The major policy which comes into focus is the requirement to use a Navy standard high order language. The UPDATE I version set as a goal to develop the Mission Software in the CMS-2 high order language. Once accomplished, the Mission Software from UPDATE I has continued to form the basis for all succeeding UPDATE Programs.

Identification and control of critical documentation is performed at the front end of the software development process. Navy documentation standards have been generated which explain fairly well the type and content of documentation required in building software. The Navy standards are covered by references (A), (B), and (C). However, management must define for their product what is acceptable as a minimum set of documentation, given cost and schedule problems, and a maximum set given the ideal situation. Once a particular documentation set is established, it is more than likely that the level selected and composition of the set will remain that way for the life of the software product. Therefore, the UPDATE I set was chosen very carefully since each document would have to prove its worth over a considerable period of time, 15 to 20 years.

The main thrust of in-house documentation efforts is producing the necessary front end requirements and interface documents, and contractor statements of work. There was a major attempt in UPDATE I to establish adequate and standard system requirements type documents. An effort was made to develop System Performance Descriptions documents for each of the subsystems undergoing a major change. Even though specifications existed for various sensor equipments, interfaces, and software elements, nowhere was there described the type of performance that was expected from a total avionics subsystem such as navigation or data processing. It was hoped that by the System Performance Description documents the Center could start to set some sort of bench marks for each of these subsystems. This would then allow measureable expected performance for present subsystem capabilities, and to better judge what would be realistic and manageable goals for improvements in the future. The four volumes created by UPDATE I covered: an UPDATE I system overview; the data processing subsystem; the navigation subsystem; and, the acoustic subsystem. These documents served the initial purpose of definitizing performance, and also greatly assisting in educating personnel on these subsystems. However, this documentation effort was not continued forward by finishing the complete avionics subsystem set or used for subsequent UPDATE versions. The needed commitment of highly specialized technical resources and long term expense being the main reasons for their demise. It must be noted that such documents still could achieve the very important roles stated above as well as definitizing more clearly contractual responsibilities between the various avionics system contractors. The fact that such documentation no longer actively exists does weaken weapon system product definition and procurement strategies.

The software functional requirements and interface design are the most important items needing definition relative to controlling the software development process. Again, UPDATE I made a large investment in producing documents for these areas. The software requirements were broken down into 20 volumes of relatively independent topics and designated the software Functional Descriptions

for the UPDATE I Mission Software. There was a great effort in these documents not to rely only on narrative form since words can have such diverse meanings to different people. So the logical flow of events/equations/interface actions between crewmembers and avionics system were also put in chart or diagram form denoting requirement flows. This helped tremendously to reduce the translation problem between ASW fleet user, software designer, programmer and tester. The second item needing clear definition is the interface between software and hardware. The interface Design Specification serves this function and is a necessity because it defines for the Software Engineer the world external to the software. Once specified the Software Engineer can proceed with some confidence knowing the demands which will be placed on the software design by the remaining portion of the data processing subsystem and the individual avionic subsystems.

The final document item requiring critical control is the individual contractor's statement of work. It is here where management edicts and controls should be stated. All the wonderful software development methodologies are somewhat meaningless if they are not backed up in the contractual statement of work. The contractor(s) should be put in such a management position as to be legally required to follow the Center's desired methodology in developing software. This is not to say that the contractors incentives and ingenuity are restricted and that they do no longer bear full responsibility for the quality of their products, because they still do. It does mean, however, the the Center designates for the contractor(s) the process, facilities and tools, software reviews and audits, critical decisions and paths, and quality, cost, and schedule controls that will be used. The emphasis on certain documentation efforts in the above paragraphs was not intended to de-emphasize the total gamut of documentation material needed for good software development, but was presented to show the most critical items used by the UPDATE versions in controlling the quality, cost, and schedule of the products.

Instituting control over the software development process is extremely important. There are several items of control which can be done technically and several items which can be done managerially. The UPDATE I technical staff specified the software product to be developed by functional "builds", that is, major functional blocks were incrementally developed and phased together, where the last "build" contained all the functions of the total Mission Software. The functions (software tasks) of each "build" were so designated as to conform to a building block concept where the base "build A" contained vital functions (executive tasks) of the system software and succeeding "builds" added more and more functional capability to the system. This "build" process allows step by step confidence in the system and presents well defined test blocks. Another control item performed by the technical staff was to develop program size (amount of memory required) and timing (amount of computer execution time) estimates from the preliminary software design information. These figures were then used to establish program size and timing budgets so that these vital life signs of the software development process can be technically monitored. The technical staff also established an embedded software performance measurement aid in the Mission Software. This item is further discussed in the Software Design section of this paper. While the technical staff concentrated on design aspects, management established Configuration Management (CM) procedures on the software process and set up configuration control boards (CCB) to oversee the CM and change functions. The primary input to this procedure was the functional description baseline established at the Preliminary Design Review (PDR) and the design baseline established at the Critical Design Review (CDR). The procedures were, and continue to be, used to control all modifications to the software after PDR/CDR's including the resolution of problems detected during software integration, system integration and formal testing. A standard change notice form has been adopted for use by all participants to help overall communications.

In UPDATE I it became important during the design and integration of certain complex algorithms, such as OMEGA navigation equations, that a second more controlled implementation be performed on a processor other than the P-3C data processing subsystem. Therefore, when such algorithms were required the set of equations would be programmed and exercised on another processor where confidence could be developed in the equations and/or logic flow. This is done by programming the algorithms on the Center's CDC 6600 Computer Facility in FORTRAN. In this manner, the critical algorithms can be validated outside the weapon system computer and the corresponding output results can be used to determine when the P-3C system is implementing the algorithms correctly, by making comparisons. Software design "tuning" is done quite extensively by comparing accuracies obtained after performing different equation programming implementations on both processors.

5. SOFTWARE DEVELOPMENT PROCESS

The P-3C UPDATE software development process is best shown in pictorial form in figure 4. The process begins by assuming that the "systems" level work has been performed properly and that system requirements and design architectural inputs are available. The software Functional Descriptions documents are then derived during the software requirements development phase. As mentioned earlier, the Center's CDC 6600 Computer Facility, is used in the validation process of certain algorithms which could prove to be difficult to implement. This computer facility is also used as required to develop the necessary equations for the various subsystems.

During this time frame a procurement package and attendant documentation is being put together for negotiating the prime software implementation contract. This package includes:

- A detailed statement of work describing the Center's software development methodology (including process, Navy standards and guidelines, and estimated man-power, schedules and cost).
- A complete set of software Functional Descriptions. Often the contractors are given a preliminary set of Functional Descriptions for generation of the proposal response because final requirements documentation are still in the process of development.
- A set of documents describing the subsystem interfaces (UPDATE III uses Interface Design Specifications).
- In UPDATE I a special contractual item was generated called the Software Design Requirements document. The document was generated by the in-house technical staff and specified for the implementation contractor the technical requirements of the Mission Software design. Such items were included as the known idiosyncrasies of the CP-901 Computer (so hopefully these inherent design flaws could be minimized), degraded software modes, system recovery modes, segmentation structure of the tasks, modules and data base, executive program attributes, analysis aides, and integration tools.

In UPDATE I, by the time the software Functional Descriptions were finalized, the implementation contract had been let and the contractor's personnel were coming on board to start software design. The Center uses the well known review tools of Preliminary Design Reviews (PDR) and Critical Design Reviews (CDR) for controlling this phase of the process. As a general rule the contractor should thoroughly understand the functional requirements to be implemented after completing PDR, and at the end of CDR the in-house technical staff should be thoroughly convinced that the proposed contractor's software design will satisfy the successful implementation of all the design and functional requirements.

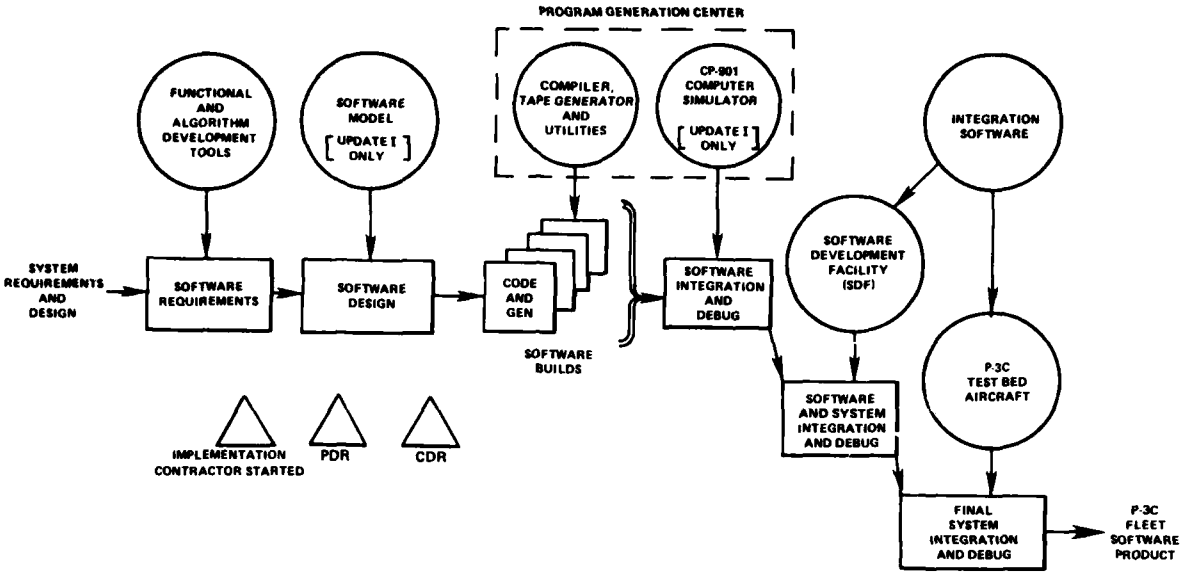


FIGURE 4 – P-3C UPDATE Software Development Process

Another tool employed during the software design phase in UPDATE I was the software Model. The Model is a software analytical tool which uses the modeling language GPSS hosted on the Center's CDC 6600 Computing Facility. The tool was used to model the contractors proposed design, and then run a set of experiments to gain confidence in this design. Major data processing subsystem elements modeled were the CP-901 Computer instruction set, the input/output commands, the proposed executive, and certain "dummy" functional application modules. The "dummy" modules represented the respective functional application modules by presenting a block of code having nearly identical size and timing requirements.

After completing CDR the software progresses into the detailed design stage by coding and generating the various specified "builds". In UPDATE I "build A" contained the most critical software tasks, and once operable was joined with succeeding "builds" which added more and more functional (applications) capabilities to the system. The compiling and tape generation for the CP-901 Computer Mission Software is performed in the Program Generation Center. The Program Generation Center's main compiling facility uses a CP-901 Computer which has been modified to operate like a UNIVAC 642-B Computer. This modification enabled the P-3C UPDATE Programs to utilize the Navy's CMS-2 tactical programming language. The CMS-2 compiler is taped oriented, however, this facility was built with disks that look like tapes to the compiler, and job throughput was greatly increased by this facility feature. The data access speed was limited by the tape speed, and this was a major factor in low facility throughput in the past. Therefore, the disks were installed because of their very high data access speed. The greatest benefit of the disk feature is the speed by which the system generator can access modules/tasks and rapidly generate the various "builds" or total system tapes. The system generator mentioned here is a software program which is hosted on the main compiling facility in the PGC and uses the disks as disks. It combines all P-3C UPDATE Mission Software components into a form suitable for a bootstrap load into the avionics system computer memory and auxiliary drum memory. The PGC main compiling facility also has a satellite system which is used extensively for building jobs for the compiler or system generator and producing hard copies of finished jobs.

The first stage of software integration and debug is performed with the CP-901 Computer Simulator. The CP-901 Computer Simulator is a software tool which is hosted in the PGC. This tool exercises selected blocks of code and provides for the programmer an indication of logical correctness of instructions and sequences. The tool also gives timing (execution) estimates for running the selected block of code, thereby allowing the programmer to assess design efficiency. The UPDATE II and III versions no longer use this tool because the modules to be tested became too large for the PGC resources. It also appeared that for large blocks of code (multiple modules) programmers would rather go directly to the Software Development Facility (SDF) because the CP-901 Computer Simulator tool is too cumbersome to handle, that is, setting the job up to perform the test runs. However, during UPDATE I this tool did allow the in-house technical staff to use the features as a clearing house for individual software modules. This information was then used to given an indication of how well the implementation contractor was technically progressing. The tool was also used to investigate troublesome areas of code.

The main stage of software integration and debug is performed in the Software Development Facility (SDF). This facility is the single largest investment in unique tools for supporting the software development of the UPDATE versions. This facility contains an actual P-3C avionics suite housed in custom built consoles, plus a complete stimulation system. Through stimulation inputs a "flying" tactical aircraft environment can be duplicated so that the software modules can be integrated and debugged. The SDF represents the prime tool in the development of the Mission Software. The SDF emulates as closely as possible, in the laboratory environment, an actual P-3C aircraft ASW weapon system. The one major exception is the physical configuration, the crew stations and avionics equipments are arranged to facilitate the building and debugging of software. The work area houses the P-3C crew station consoles and stimulation control console. This area has a working environment which is conducive to technical development. Lights are dimable and the equipment noise is at a low level. The climatic conditions (temperature and humidity) are easily controlled, and the interior decor compliments the work being accomplished. The mainstay of the P-3C avionics suite (rack mounted equipment) is housed in the avionics room, adjacent to the SDF proper. This is done so that the heat and noise generated by the equipment will not interfere with the environmental conditions of the working area. Also, the avionics room contains those equipments which require no operator intervention. The equipment in the avionics room is mounted on racks which are designed for easy maintenance and accessibility. The separately controlled climatic conditions of the avionics room also greatly improves the reliability of aircraft equipments, thus giving a high degree of facility availability.

The design objective of the SDF is that system software development and verification be performed to such an extent in this facility that only a minimum number of hours of actual testbed aircraft usage will be required in the final development and verification phase. This facility presents a sterile environment for performing software development in support of all the P-3C UPDATE versions. The SDF can be easily reconfigured (switchable) into any of the existing P-3C UPDATE versions, thereby, multiple teams of users can be accommodated in a very cost-effective manner. Also, the software development efficiency presented by the SDF has greatly reduced the requirement for aircraft ground integration and debug of software, and the corresponding flight testing to verify operability. Tremendous confidence can be gained in the design of the system software before the Center need commit these versions to flight test and acceptance.

There is an important class of software called Integration Software which should be mentioned at this phase in the process. Integration Software is developed for validation of the new equipments which are added during each UPDATE version. The first function of the software is to determine if the new equipments input and output signals are in conformance with the individual equipment specifications. The second function performed is to integrate and validate installation of the various new equipments as they are placed in the SDF and P-3C Testbed Aircraft. This is an important step since users of these facilities need to have a high degree of confidence in the soundness of the tools and equipments before using them to develop their software.

The final stage of the system integration and debug phase is performed in a P-3C Testbed Aircraft. The aircraft is equipped with an UPDATE version avionic suite, and flight testing is performed to complete the cycle of total system software validation against the complete set of software Functional Descriptions. The P-3C Testbed Aircraft is also used by a Navy designated agency for acceptance testing of the Mission Software. This agency's task is to independently assess the functional performance and design quality of the Center's product. At the conclusion of flight testing, and after Navy acceptance, the Center delivers the UPDATE Mission Software product to the Fleet.

The above described process is designed to take the software through three levels of progressive testing: modular software design testing level (CP-901 Computer Simulator); partial system software design and functional testing level (SDF); and finally, total system software testing level (P-3C Testbed Aircraft). There are test procedures and flight scenarios which have been developed covering the total gamut of testing presented by this development process. Each set of tests must be passed (clearing house) before proceeding to the next stage. The process therefore allows management strong visibility and quality control over the product.

Two items which have not yet been discussed but perform a vital role in the process are the Automatic Documentation Center (ADC) and the P-3C UPDATE Library. These facilities play a major role in the Center's task to keep pace with the documentation generation for software programs of this size. The ADC consists of communication terminals for data input and a high speed printer for documentation generation, all located at the Center. These equipments are linked via telephone lines to another Government computer center. The software program on this computer performs text editing, recording and retrieval operations. This capability of using data processing to store and manipulate the documentation data base has greatly enabled rapid and economic changes to be made to the documentation. The P-3C UPDATE Library has served as the single control point for the storage of software documentation and distribution of software deliverables. Additionally it stores a significant amount of P-3C technical data covering aircraft, aircraft support systems, and avionics system items for the basic P-3C and the three UPDATE versions. Both facilities have been invaluable in controlling the enormous amount of documentation generated in support of the UPDATE software versions.

6. SOFTWARE DESIGN

The software design topics discussed in this section will reflect the technical items of concern relative to managing the P-3C UPDATE software developments. The software design of the initial 1969 fleet issued program was: structured to be core resident; a fine-tuned assembly language program; a non-structured design by today's standards; and, the program was at memory capacity of core so functional changes could only be made by removing functions or by more efficiently designing existing code. These factors made software changes extremely difficult. For the above reasons, plus the system requirement to add the drum memory, the decision was made that UPDATE I Mission Software to be completely new start, incorporating more advanced and innovative features in the software design architecture.

To provide guidance in the process of developing a new architecture, a list of the major features desired of the UPDATE I Mission Software was generated. This list was compiled by noting the problems being observed with the existing fleet issue program and considering the design base required to accommodate the building block concept instituted under the UPDATE series. The list is presented here for general understanding of the objectives sought by the UPDATE Program:

- The software should exhibit good stability, and should operate without fault over a complete mission (15 hours).
- The design should be structured to make optimum use of data processing system memory and execution power resources.
- The software should possess the capability of degraded modes.
- The design should be structured to allow major changes to be performed without any redesigning of the basic architecture, that is, the easy addition or deletion of software tasks and modules.
- The structure and nomenclature scheme used for the software elements should be so organized as to facilitate easy Configuration Management.
- The software functional and design documentation should be of such quality as to allow the same software to be repaired by an agency other than the developer.
- The design should be structured by utilizing well defined logic paths, linkages, data base organization, and debug features so as to make possible easy software fault isolation and repair.
- In general, the design should be modularized to the maximum extent possible. Tailored or optimized design should only be permitted when it is clear that the efficiency of a particular block of code needs improvement to operate properly.

The above stated objectives and the in-house proposed software design architectural features were written into the UPDATE I Software Design Requirements document mentioned earlier, and formed the basis of the software design guidelines for the implementation contractor. A pictorial of some of the software design architectural features is presented in figure 5. The UPDATE I Mission Software was coded to the maximum extent possible in the Navy's CMS-2 Tactical Programming Language – a standard high order language. Software elements like the executive, communications link, input and output sequences, and timing loops were coded in assembly language because of critical timing requirements and execution time efficiency.

Figure 5 shows four architectural features which have given the software flexibility in design and made future modifications much easier. These features are the standard interface, common executive, centralized input and output to avionics devices, and module independence. All software requirements are organized by having major functions as "modules" (5000 - 20,000 instruction words) and subinstruction sets as "tasks" (50 - 500 instruction words). The range of the number of instruction words for modules and tasks is presented to illustrate the approximate relative organization of the design structure and is not intended to depict the maximum and minimum number of words used in each type of software element. The design was structured into as small as practically possible logical "tasks". When specific "tasks" are integrated together they form software "modules". When all the "modules" are integrated together they form the UPDATE Mission Software. The function of any one module closely supports one of the avionics subsystems such as acoustics or navigation. New modules can be added or deleted from the system by linking into the standard interface as shown in figure 5. The module being changed carries its own data base and little if any modifications are needed to the remainder of the Mission Software. However, if the module being changed is accompanied by hardware, then individual task changes must also be added in the input/output and resource allocation portions of the executive module so proper timing, priorities and communications can take place in support of the changed module.

Three other features of the software design should also be mentioned because of the vital role they perform in system recovery and measuring software performance. The software has designed into the basic structure a series of "flags" that will set themselves whenever there is a detected software failure. The "flags" are designed to respond to parity errors, incorrect timing and so on. The crewmen are alerted to these problems by being presented with visual cues on a display. This failure reporting feature is one item that has helped build confidence in the system. A major feature which is in direct support of system recovery is degraded mode. Because the executive program has been designed with a dynamic program allocation scheme, that is, real time switching of tasks/modules between core and drum memory, it can also be used to reconfigure the system when certain components are not operating correctly. However, the reconfiguration is not done automatically, but is performed under operator control during an execution of a system restart. The final feature to be discussed is the analysis aide. This is a special information gathering routine built into the Mission Software to allow data to be collected during flight testing. The routines were designed to collect such information as execution time utilization, number of input and output requests, and which modules/tasks requested servicing the most. The information obtained from the flights is brought back to the laboratory and used to measure the performance of the Mission Software. The analysis aide feature is not delivered to the Fleet as part of the Mission Software but rather used only during development.

The Mission Software design is documented in general accordance with the requirements of reference (A). The documentation set for an UPDATE version includes Computer Performance Design Specifications (CPDS), Computer Software Design Documents (CSDD), a Computer Data Base Design Document (CDBDD), a Computer Program Test Plan (CPTPL), Computer Program Test Procedures (CPTPR), Computer Program Operator Manuals (CPOM), and a Computer Program Package (CPP). Particular emphasis has been placed on block diagrams, program flows with narratives, and cross correlation between narratives/documents and listings. Some exceptions have been taken with this prescribed set to provide the most cost effective documentation, considering both documentation production costs and ease of program maintenance. The primary concern with the reference (A) prescribed structure was the cost effectiveness of low level (that is, approaching instruction level) documentation other than well annotated listings (design comments in narrative form printed right on the listings), and the continued need to upgrade material to reflect the "as built" status as opposed to letting it remain in the "as initially designed" status.

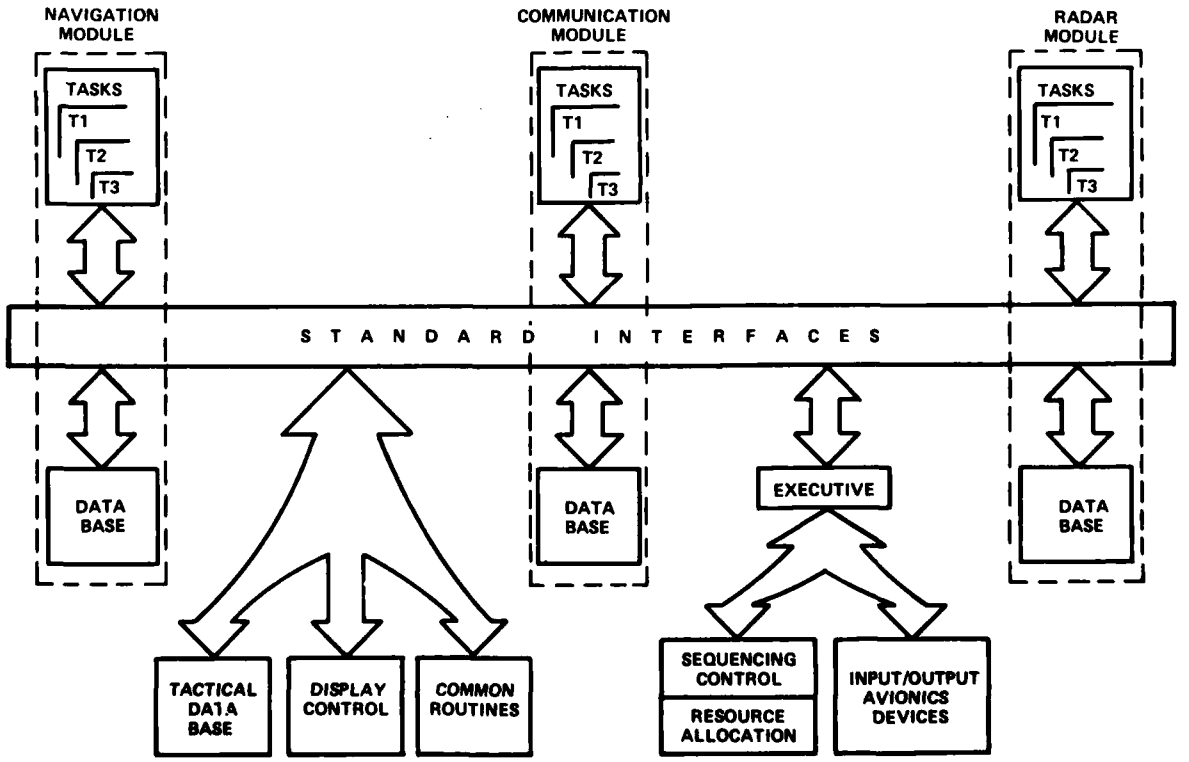


FIGURE 5 – Software Design Architecture

The documentation for the UPDATE Mission Software was designed to satisfy the following general guidelines:

- Sufficient information shall be provided to clearly describe all aspects of the software development process: that is, functional and performance requirements, design and implementation descriptions, test plans and procedures, and user and maintenance manuals.
- Sufficient narrative and detail shall be provided to permit someone other than the original developer to maintain the software. Narrative shall be easily correlatable with program listing comments.
- Repetition of system and programming information in multiple documents shall be avoided whenever possible. In UPDATE I system descriptions and often used programming information presently incorporated in several reference (A) documents were consolidated into a System Programmer's Reference Manual. This document was then referenced as necessary in the appropriate areas within the reference (A) documentation tree.
- A standardized listing notation procedure shall be defined and utilized throughout the whole software development process.
- The Automatic Documentation Center shall store most of the documents generated during the Mission Software development. Whenever possible these shall be used as the baseline for documentation of later fleet issued programs.

The above discussion on software design and documentation deliberately stressed those items which are of special interest in managing software development. There are many other technical items which could be discussed in this section such as software patching philosophy and control, software library organization and control, nomenclature standardization, and Configuration Management procedures. This paper was not intended to cover all areas, and eventually these topics and others must be considered in carrying out successful software developments.

7. CONTRACTING STRATEGY

The utilization of the in-house technical staff is critical to any contracting strategy and will likewise be discussed along with the contractor tasking. The strategy depicted here is an outgrowth of the before mentioned management objectives and approach, and as will be seen naturally blends with the UPDATE software development process. A pictorial of the strategy is presented as figure 6. The following strategy was adopted for the UPDATE I Program and some modifications have been made by subsequent UPDATE Programs. However, the depicted strategy is still representative of the procurement approach used today.

During the system design and software requirement generation phases it is paramount to a successful Center software development effort to have very strong involvement by members of the in-house technical staff. It is during these two phases in the process when the requirements (what you are buying), architectural design (framework for the product), and development method (how you are going to accomplish this product development) are specified, and the Center as the prime developer must make its management role and product requirements known to industry. Therefore, the system design phase is primarily done in-house. The Center seeks help in performing the tasks by complementing its staff with assistance contractors. The term "assistance" is used to denote those contractors performing tasks in the development process other than the prime implementation contractor. These contractors generally do not have "final" authority over specified end items (such as system requirements documents and UPDATE facilities), but do have full responsibility for operating facilities, generating technical reports, and developing sections or preliminary versions of specified end products. Therefore, they do bare full accountability for specifically defined items in their respective contracts. These are not "services" contracts. Each contractor under the subject strategy has definitized and visible end products.

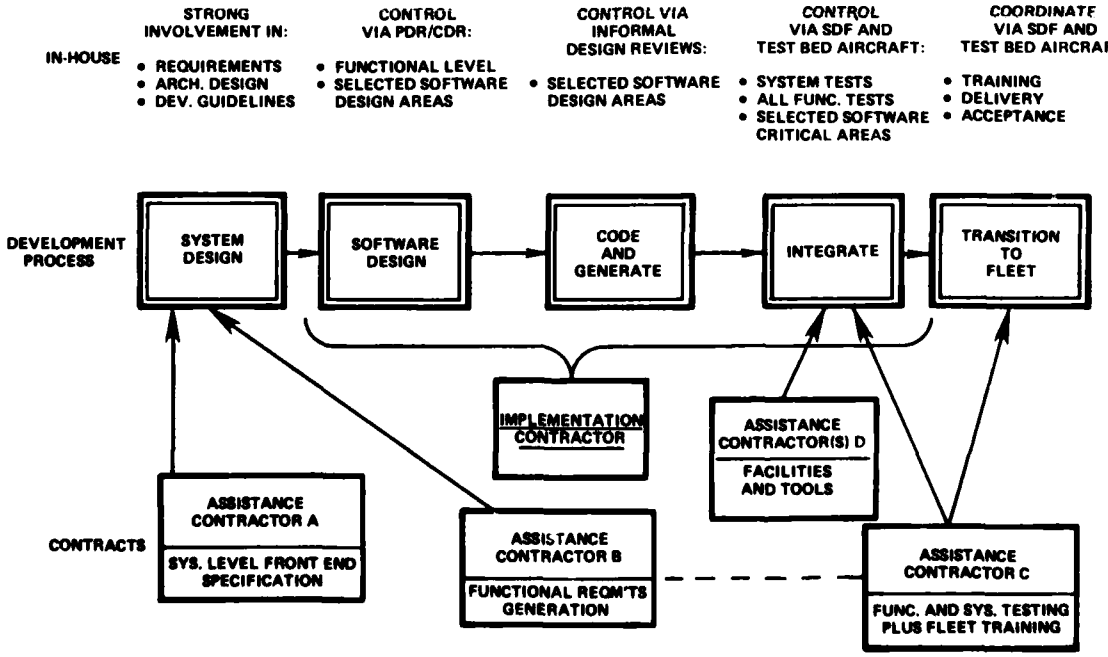


FIGURE 6 – Contracting Strategy

The system design phase is performed with the help of Contractors "A" and "B". The assistance tasks performed by contractor "A" encompass developing system level requirements and postulating a system architecture. The in-house staff during this phase finalizes these items. They also write the software development process and guidelines into statements of work for the remaining contracting efforts, including the implementation contractor. Contractor "B" during this phase has a much larger, in terms of task size not criticality, role to perform. They must assist in the development of all the software functional requirements. This effort includes collection of the necessary technical background data, and generate from that data the UPDATE Mission Software Functional Descriptions. This set of documents then constitutes the product description and will be used as the binding legal requirements in the implementation contract. It should be noted at this point that it is the Center's desire not to have any of the assistance contractors bid or participate as the Mission Software implementation contractor. In this way the Center maintains good checks and balances over the process, and keeps better control over cost and schedule. After contractor "B" assists in writing the software requirements they are in a position to be most profitably employed as assistance contractor "C" for writing the test plans and procedures; the task for validating conformance of the product with specified requirements. It can be seen that once contractor "B" has assisted in generating all the Functional Description documents they are then highly qualified to write individual functional tests, without having to know about actual software design implementation.

The Center performs a lead role in the software implementation phase because of the strong in-house software engineering staff which has been developed over the years. This staff monitors the quality and timeliness of the product as it proceeds from software design through system integration. They employ all of the tools and management techniques discussed in earlier sections. However, it is noteworthy to mention that the in-house staff concentrates on selected software design modules/tasks which were identified earlier in the system design phase as new and/or critical to the success of the UPDATE version(s). In the case of UPDATE I for example, the staff closely monitored the work performed on the executive control module because this was by far the most critical building block of the whole UPDATE Mission Software. A more recent area of in-house concentration is the work being performed in UPDATE III on the Advanced Signal Processor software. Since this is a new module being added to an existing base it is therefore more cost effective to concentrate in-house resources in this area.

Having produced the system level documents, key statement of work and software Functional Descriptions the strategy is now to let a contract for the implementation of the Mission Software. The term implementation being defined as the design, code and generation, and integration phases of the software development process. This contract is obviously the largest in the subject strategy and requires considerable continued evaluation of the contractor's performance for quality, cost and schedule. The contract during UPDATE I version was let competitively to promote a corporate commitment from one of the larger American industrial software houses, although all industrial bidders were welcome to participate. This strategy was realized and a new contractor was brought on board at that time, circa 1972, to implement the UPDATE I Mission Software under the Center's leadership. Continued managerial and technical monitoring of this contractors work is a very important Center task which must be performed to insure that the key elements of this subject contracting strategy are being executed in a cost effective manner.

The Center's approach, as stated previously, is to own and operate all of the facilities and software support tools necessary to carry out the software development process. Although the support items are physically located in-house the contracting strategy is to have, as much as possible, their development and operation performed by assistance contractors "D". The Center's in-house technical staff manage the development of these unique facilities and tools, letting contracts in such a manner as to stimulate innovative ideas for these devices and fixing responsibility for certain support products with specific contractors. Likewise the Center oversees the operations of these support items, but lets contracts for personnel to actually run the operations of the facilities. All of these contracts are let in the light of maintaining checks and balances for good software management.

The final phase of the process, transitioning the products to the fleet, continues to be performed with a balance of responsibilities between the in-house staff and contractors. Because Contractor "C" helped develop the software requirements and performed testing to check conformance with those requirements they are again the most knowledgeable assistance contractor to help in performing: the training of the Navy acceptance team and Fleet personnel; assist during the independent Navy acceptance testing of the Mission Software; and, assist in delivering software packages (tapes and documentation) to the Fleet.

A general management guideline in distribution of personnel resources is presented here to put into perspective the balance between in-house and contractor work forces. The UPDATE I Program's personnel resource allocation will be presented as an actual example. Approximately 25% of the Program's personnel resources were in-house and the remaining 75% supplied by contract. The contract percentage is becoming slightly higher with subsequent UPDATE versions.

8. LESSONS LEARNED

This section will elaborate on some of the lessons learned while trying to manage and technically control the UPDATE I software development effort. Brief explanations will be presented on the items of the UPDATE I approach and process which encountered problems, along with suggestions in some areas on how the problems might be remedied for subsequent UPDATE Programs.

The software Model, although technically a sound approach to validating proposed software designs, proved to be untimely in the UPDATE I Program development schedule. The Model did provide good technical design information once developed. However, because of the overall development schedule the Model work was overtaken by the need to commit the proposed implementation contractor's design to coding. The lesson learned is that in order to use this tool effectively the modeling task must be started well in advance of the implementation phase. Also, management must be willing to commit to the expense of keeping the Model current with the actual system design – this can prove expensive in both cost and technical personnel resources. There was also a contractual strategy mistake made relative to the Model. Management failed to include the generation of input data to the Model as a contractual deliverable in appropriate contracts. The UPDATE I Program tried to rely on informal data deliverables and the corresponding delays added to the untimeliness of the Model work.

The largest single mistake of the UPDATE I approach was not following through on tracking the software program memory size and timing budgets in such a manner as to control these two critical items of the system. The budgets were generated from design estimates during the software design phase of the development process. As a result of this failure, both items were up against resource limits at the completion of the UPDATE I development process, leaving no system software resources for the very much intended future UPDATE Programs. The UPDATE II version redesigned and reprogrammed many areas of the UPDATE I Mission Software to rectify this problem. The lesson learned was that the in-house staff must devise methods, either manually or hopefully automatically, to technically monitor these vital life signs in the software development process. It should be noted in fairness that the UPDATE I Program did have an accelerated schedule (which was met) and the resultant design and code efficiency for memory and timing resources suffered because of this factor.

Another tool which developed problems was the CP-901 Computer Simulator. The problem with this tool was mentioned earlier in that it was limited by the size of modules/tasks it could test because of the host PGC resource limitations. Since many of the UPDATE I software modules grew quite large in size, the use of this tool became prohibited. The lesson learned is that when building support tools, development should be on a host system (computer facility) that has the size, power, and features to allow for extensive growth. New and advanced software support tools being developed today by the Center such as the Facility for Automatic Software Production (FASP), reference (D), are hosted on the Center's CDC 6600 Computer Facility. The FASP offers modern software engineering tools which are presently being utilized by the UPDATE III Program for development of the acoustic subsystem software.

One problem which might be obvious was the difficulty in managing and negotiating, and continuing to renew, all the contracts brought about by the stated strategy. There is the problem of generating legally correct contract packages and having them negotiated with such timing as to reap maximum benefit from the contracting strategy. There is the second problem of defining correctly the legal interaction between all contractors/agencies in the development process. That is, the contracts must judiciously define the individual responsibilities of each contractor and clearly specify the Government Furnished Information (GFI)/Government Furnished Equipment (GFE) and Contractor Furnished Information (CFI)/Contractor Furnished Equipment (CFE) for all contractor/Center interactions. Failure of any one contractor to produce items or information on schedule can cause multiple problems for management. The lesson learned is that management must be willing to commit a major portion of their in-house resources to managing this contracting strategy. The strategy and process as defined in this paper continues to be used today, and is considered a successful approach to managing the large P-3C UPDATE software development efforts.

Another contractual aspect overlooked in UPDATE I was the desirability of having selected "builds", developed by the implementation contractor, delivered as contract line items. Although the informal deliveries did not necessarily hinder the UPDATE I Program it is seen as good management practice to specify these major software elements as contract obligations. The chief reason being that in the event design difficulties are detected early in the "build" cycle Center management can take formal action in requiring the design problems to be fixed, rather than relying on informal requests or waiting until the final product delivery to make corrections.

The final problem to be discussed is one which developed after the UPDATE I era. This problem came about because of the evolutionary nature of the program which developed multiple configurations of UPDATE Mission Software. Naturally there must be different versions of the Mission Software since the various UPDATE capabilities, which are also hardware related, were not backfitted into earlier versions of the avionics system. However, because of critical timing between the various UPDATE development programs, there was a rapid drifting from a common baseline configuration. There are management actions being taken today to bring about much stronger commonality (common modules/tasks) wherever possible. The lesson learned was that in the real world of building multiple and evolutionary product lines, it is extremely difficult to control commonality. Major resources must be commited early for the product line development in the areas of Configuration Management. Although Configuration Management was done quite successfully for individual Mission Software products, mechanisms need to be developed and enforced to control developments in the multiple product environment.

9. CONCLUSION

It is hoped that the reader has gained some understanding of the Center's P-3C UPDATE software development methodology. In summarizing, it would be helpful to present a pictorial of the major control points and critical items that the P-3C UPDATE Programs use in gaining assessment of the "health" of a particular software version as it proceeds through the development process. Figure 7 presents the pictorial of this information. It will not be necessary at this point to present explanations of these control points and critical items since all the items have been addressed in some detail previously. If a software development proceeds past a control point without successfully fulfilling the requirements of a critical item, there is a high probability that the program development is going to face difficulties.

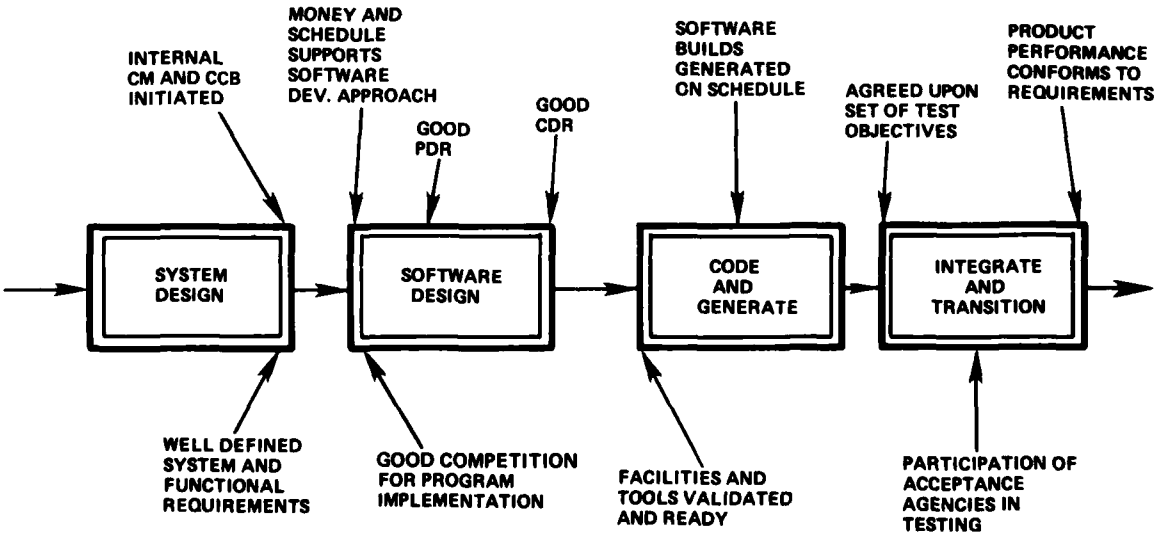


FIGURE 7 – Software Development Control Points

It is important to note that management practices surrounding any methodology must remember that software development is a people driven process. Management must be aware of the fact that success in this business depends on the ability to constantly generate a team spirit – all team members must know and relate to a common goal and process. Management must know what motivates software engineers and programmers and use this knowledge effectively. The quality of P-3C UPDATE software products ultimately rests with the Center's professional managers and technical staff. This staff has been successfully utilizing this software development methodology since 1972.

10. REFERENCES

(A) Requirements for Digital Computer Program Documentation, Weapons Specification WS-8506, U.S. Department of the Navy, Naval Ordnance Systems Command, of November 1, 1971.

(B) Tactical Digital Systems Documentation Standards, Secretary of the Navy Instruction, SECNAVINST 3560.1, U.S. Department of the Navy, of August 8, 1974.

(C) Weapon System Software Development, Military Standard, MIL-STD-1679 (NAVY), U.S. Department of Defense, of December 1, 1978.

(D) Mr. H.G. Stuebing, A Modern Facility for Software Production and Maintenance, Guidance and Control Panel AGARDograph on Guidance and Control Software, of 1979.

Executive Software Reusability for Distributed Avionics Architectures

by
R. F. Bousley
Manager, Advanced Avionics Software
Military Airplane Development
Boeing Aerospace Company
P.O. Box 3999
Seattle, Washington
98124
U.S.A.

SUMMARY

With the current capabilities of microprocessors approaching that of minicomputers and the reliability of these small processors increasing, the avionics system designer has an opportunity to restructure his system to allow processing at the point where data is collected, and then transmit or receive data of a system nature over a global data bus. The consensus of opinion at the present time is that the microprocessor will allow more distributed networks such as hierarchical networks with functional avionic groups (NAV, COMM, weapons delivery, stores) isolated on local, lower level data buses, to be used effectively in future avionics systems.

What impact these distributed microprocessor-based systems of the future will have on integrated (mission-oriented) avionics software is a major concern to the system designer. It appears that a functional decomposition of the executive software will help to maximize the reusability of this software since only those functional modules of the executive software which perform hardware dependent functions will need to be changed. A modularized executive, along with a firm executive/applications software interface, will allow maximum reusability of the applications software. This will minimize the impact on the integration task during mission software development.

1.0 BACKGROUND - AVIONICS NETWORK EVOLUTION

Many years ago the pilot was the central processor and computational unit in the avionics system. Obviously much of his time was required to assimilate all the available data. The pilot was often distracted in the decision making process, because he was primarily controlling the aircraft. It is unthinkable to expect the pilot to collect and assimilate data without large processing systems in this day of highly sophisticated, high performance aircraft.

The centralized system, controlled by a single large scale central computer, was developed approximately 15 years ago and has been the main integrator of avionics. The centralized processing system was definitely an improvement over earlier systems, since complex data and control was processed and presented in simplistic form for the pilot. However, there were inherent problems in early centralized systems; computers had small computational capability, primarily due to cost, size and weight constraints, and power and cooling requirements. Additionally, the software developed to operate in an avionics system of this type was inflexible and growth was difficult to accommodate. Another problem was that of reliability, with all subsystems connected to a single control point. This problem was solved by adding a redundant central processor. The problems of cost, size and weight, power and cooling were then even greater. However, the system was again a major improvement over previous systems, because information could be processed for more effective utilization. The development of the militarized minicomputer helped to alleviate some of the problems in the centralized systems, primarily those problems associated with the physical installation of the system on the aircraft. The software and system organization still proved to be inflexible and possessed limited growth capability.

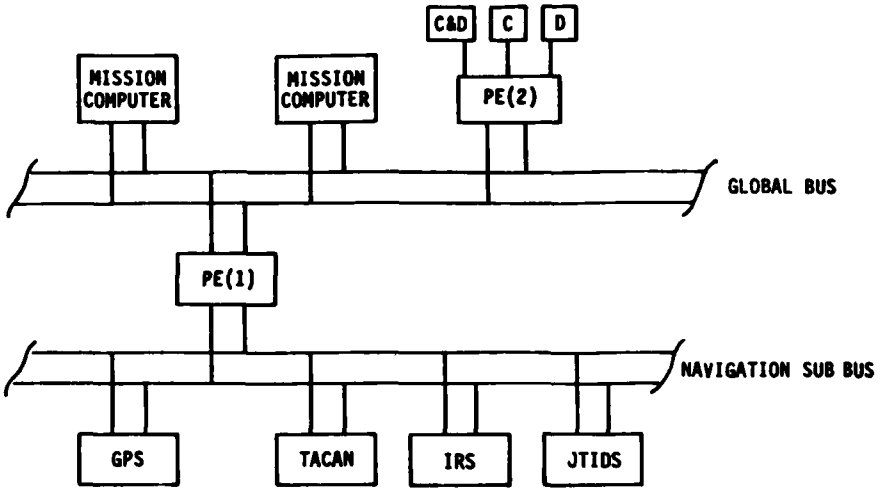
In the early 1970's, the United States Air Force Avionics Laboratory (AFAL, WPAFB, Ohio) embarked on a program designed to address those problems associated with centralized control systems. The major thrust was the DAIS program in which a number of standard minicomputers were distributed via a MIL-STD-1553 multiplex data bus. This type of system allowed the processing capability to grow, increased processing capability since processing could be accomplished in multiple computers and allowed a greater degree of modularity at a processor level. The system reliability was increased, since the functions were distributed.

It is widely believed in the military and industry that the microcomputer will permit more of the processing to be distributed, that the processing networks will be an extension of the DAIS-type distributed system; processing for subsystems will be distributed on local multiplex buses and these hierarchical processing networks will communicate only system data over a global bus. This approach allows subsystems to

have greater flexibility and each subsystem can be tailored to meet its specific mission requirements. Since the data used by the subsystem is transmitted on the local data bus, system level software need not be changed to accommodate a change to the subsystem, because the data presented to the global bus for a subsystem will be preprocessed and not necessarily reflect characteristics of unit level subsystem components.

In a hierarchical network (figure 1), failures at the local level can be detected and handled at the local level where the most knowledge of this failure impact is understood. For example, the mission computer programs need not be concerned with a failure of one of the elements on a local bus if their computational algorithms are not dependent upon the source of the data as much as the integrated data on the global bus.

With a hierarchical network, a number of possible architectures are made available. For this discussion architecture is defined as a combination of the network (hardware elements and interconnections) and the bus control mechanism (the control of the data flow between elements in the network). Networks and bus control mechanization issues are discussed in the following paragraphs.



HIERARCHICAL SYSTEM EXAMPLE WITH NAVIGATION SUBSYSTEM ON LOCAL BUS

NOTE: All integration of the navigation data occurs in PE(1) and only integrated navigation is passed to the global bus (to the maximum feasible extent).

FIGURE 1

2.0 DISTRIBUTED CONTROL SCHEMES

Control scheme or mechanism is the method employed to determine which processing element in the network has control of the data transfer medium (for example, MIL-STD-1553 data bus). The available control mechanisms can be categorized into two general types, either stationary or nonstationary. The major difference between the two is whether or not the bus control capability is passed from one location to another or maintained in one location.

Stationary Bus Control

In this mechanism there is only one bus controller per bus or bus pair (active, standby). The only exception to this is the case where the single bus controller fails and a monitor (or backup) bus controller takes over and continues control of the data bus.

This control mechanism has proven itself in aircraft applications (F16, F18), but is vulnerable to system point failure and must be implemented in a redundant manner to prevent low system reliability. Also, since all bus control is isolated to a single point bus control tends to become increasingly complex and inflexible due to this complexity.

Nonstationary Bus Control

Although the stationary single point control mechanism may well have application in future avionic systems, it appears that this mechanism may be replaced by a

nonstationary multipoint control scheme. There are a number of different nonstationary control mechanisms defined and being discussed today for future use. The primary difference between these proposed schemes is the mechanics of determining which processing element is to get control of the bus next. Some of the leading and most likely candidates for bus control mechanisms for future avionics systems which are:

- o Round Robin
- o Polling
- o Contention

All three of the control mechanisms are alike in some ways. For example the bus controller function exists in a number of places, but the physical bus controller does not move. However, each bus controller has the ability to be the next bus controller or master. Each bus control mechanism contains a bus acquisition functional element which will aquire control of the bus. Also, each bus control mechanism has a bus control handover functional element. Typically, regardless of the operating principle used in the bus control mechanism, the amount of bus control each controller is allowed is a transmission session which may be a message sequence or a fixed time.

Round Robin

This distributed control mechanism is a Newhall loop where control of the bus is passed in some predetermined manner, such as passing the master bus control from a bus access list or table. When a bus controller finishes a transmission session, then control of the bus is given up or handed over to the next potential bus controller. If the next potential bus controller has need for control of the bus, then a transmission session is started, otherwise control is passed to the next potential bus controller and so on around the network.

The major disadvantage to the round-robin distributed control mechanism is that the system design directly impacts the bus access list format. Any change in the system must be reflected in the bus access list to accommodate any new or deleted system processing elements. Additionally, a failure in the chain requires some level of sophisticated error detection capability at the system level. Error handling considerations become more difficult to mechanize if any of the bus access mechanism (such as next address in chain) is included in hardware.

Polling

In this bus control mechanism, the bus controller will poll all other potential bus controllers for a positive response when a bus controller has completed a transmission session. In the event of all negative responses, the current controller may be allowed another transmission session, or the system design may dictate that the controller continue to poll until a positive response is received.

One of the major advantages of this type of bus control mechanism over the round-robin is that the long delay times usually associated with round-robin bus control schemes are reduced at the cost of greater overhead for control transfer (1). Other advantages include greater system flexibility, since a multimission processor could, when activated in response to a poll, initiate all of the data transfers necessary to support its roll in the mission.

Contention

The contention bus control mechanism differs from round-robin and polling primarily in the way the next bus controller demands or contends for the control of the bus. This could be considered similar to a positive response to a poll, with the difference being that response to a poll is an answer and not a demand and must be analyzed as such. A system of this sort will be data source oriented meaning that a processing resource with data available for distribution (either hardware or software) will initiate a demand for bus control in order to route the data to the proper destination. With this type of control mechanism, a demand for bus control must occur during a quiet (nonbusy) bus period. If the bus is busy, then the bus acquisition functional unit will continue to test for a nonbusy bus. In the instance where the bus is busy, a mechanism in the controller is necessary to prevent continued collisions of data. Smith and Crossgrove et. al. suggest a methodology (2) in which each potential controller after finding a busy condition delay for a random number of time units and try again. If upon the subsequent attempt the bus resource is nonbusy, it is assumed that the resource is available for use. Whatever algorithm is employed, it is apparent that some delay mechanism be employed in the bus seizure to prevent a possible proliferation of collisions (a collision being defined as two bus controllers attempting to seize the bus at the same time).

3.0 EXECUTIVE SOFTWARE CONSIDERATIONS

EXECUTIVE MODULARITY KEY TO GROWTH AND ADAPTABILITY

There has been concern expressed relative to the impact upon executive computer program design due to distributed architectures. Much of this concern is well-founded

considering some of the executive programming disasters of the past. It is the opinion of the author that the potential solution to the executive software problems of the future is neither new or particularly unique. The solution is to keep the software as simple as possible. There are two keys to this methodology:

- o Functional modularity
- o Implement only necessary functions

Functional modularity is not particularly new but it is not employed often with realtime executive programs. However, more recent developments have indicated that the concept of a functionally modular executive is being adopted.

"Implement only necessary functions" refers to the practice of overkill when defining and designing the functions the executive software must perform. This has in the past ranged from just including functions "that have always been there" to the inclusion of functions that are not necessary and perhaps even detrimental to overall system operation.

It is believed that a modular executive software design can be heuristically constructed and readily modified to accommodate various architectures as the avionics system dictates. The remaining portion of this paper addresses a modular executive design that can be adapted to the many architectures (combinations of networks and bus control schemes) which can be constructed for avionics systems.

An executive computer program has very few basic functions to perform. These functions include a hardware interface, data control and applications service. As one examines these functions they can be considered sets of subfunctions. An example is the data control function which can be broken down into queue management, and an interface with the hardware interface functions. Moreover, if one looks at or examines many executive programs, a great deal of similarity, even repetition or duplication, becomes apparent. When executive software operates in a distributed architecture some new functions are added, but the executive programs are still functionally similar.

In order for a modular executive approach to be workable there must be a well defined and enforced interface between the executive and the applications computer programs. An example of this type of interface which appears to be workable, although perhaps not optimal, is the interface definition generated by the USAF DAIS program and documented in DAIS publication SA 201307. Additionally, there must be a rigidly defined and enforced functional definition of the modules within the executive program as well as the interfaces between these modules.

An executive for an avionics application can first be thought of as consisting of two major functions. The first is bus control, and the second is local control, which is responsible for those executive functions which are local to a processing element. These local control functions include data control, task control, applications executive service, etc. Functionally then, an avionics executive computer program appears as:

FIGURE 2

Bus Control

The bus control can further be defined as a bus controller function, a bus interface function, a bus error handling function, a system synchronization function and a system status monitor function. Two of these modules are dependent upon whether they reside in a controlling processor (MASTER) or a noncontrolling processor (REMOTE). Thus the avionics executive can further be broken down as:

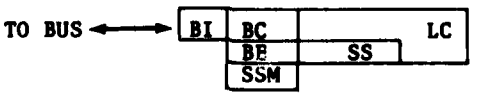


FIGURE 3

Bus Controller Master (BC Master)

Functionally the bus controller does just exactly that; it controls the bus. Usually the bus controller will be activated by a request from a local control function. Then after ensuring that acquisition of control is complete the required bus transmission will be started. In some cases, such as in the stationary master concept, bus acquisition is a function of system initialization. The bus controller

function is also responsible for data reception which is primarily queue management or perhaps, depending upon the bus interface hardware, may involve the setting of control registers. This is accomplished via an interface with a bus interface function.

Bus Controller Remote (bc Remote)

This bus controller module operates in a processor that is participating in bus transmission but is not an active bus controller (e.g., master). The functions performed by this module are minimal and primarily consist of requesting the setting of hardware registers in a bus interface unit (BIU) to ensure proper data transfer between the host processing element and the data bus. Another function of the bus controller is to prevent the bus from accessing data while it is being updated by software in the host processing element.

Bus Interface Master (BI Master)

The bus interface module does the actual interface with the bus hardware upon request from the host via the bus controller software. This may include the functions of bus acquisition, bus handover, bus acquisition/handover, bus interface hardware register manipulation for data transfer/reception, and the fielding and saving of status data from data sources or internal bus interface hardware for the purpose of bus error handling or the bus controller. This module is dependent upon the actual hardware employed, the network selected and the bus control scheme used in the avionics system.

Bus Interface Remote (bi Remote)

This module would provide for a remote executive (nonmaster) the same functions as the BI module does for a master or bus controller executive. These functions include the fielding and handling of interrupts. Another function would be the setting of hardware registers upon request from the bc module. Generally there would be no bus acquisition or bus error handling because there is no bus control.

Bus Error (BE)

The bus error module is responsible for isolation and responding to errors detected in bus transmissions. This capability may include only simple message verification or sophisticated avionic error handling. Bus errors are isolated and notification of these errors are communicated to the bus controller or system status monitor to take the prescribed system action.

System Synchronization (SS)

Responsiblity for system synchronization is provided by this functional module. This may include minor cycle synchronization, time synchronization or event (such as mode) synchronization. This functional module interfaces with the bus controller function to provide whatever synchronization messages are required in the system.

System Status Monitor (SSM)

This functional module would tally errors for and maintain status of operational equipment and would interface with bus error handler and bus controller to maintain system status.

Local Control

The local control function can be further defined, to include the local controller function, the executive service function, the data control function, the local error handler processor interface and the local I/O function.

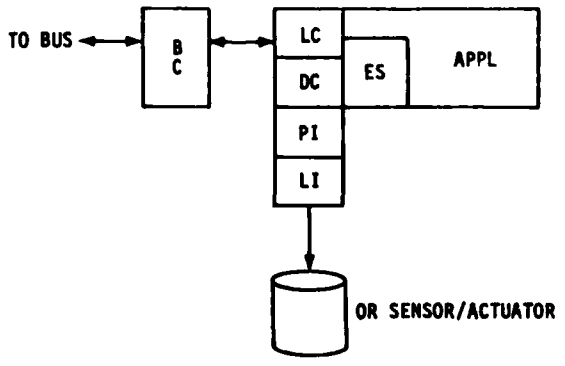


FIGURE 4

Local Controller (LC)

The local controller functional module controls the execution of all executive functions within the executive as well as those functions within the applications tasks. Whenever a task completes execution, or the handling of an executive service changes the state of the local processing environment (e.g., task suspension due to wait), or the receipt of data from an external source, the local controller is activated to determine what to do next. For example, the receipt of a system synchronization message from the current bus controller will cause the local controller to be activated and the data control function will set up any queue pointers or BIU control words required.

Executive Service (ES)

The executive service function consists of a set of modules that provide executive services to the applications programs. The number of modules within the ES is system dependent. These modules collectively provide one of the key aspects to modular development of avionics software, which is to provide one interface between the applications program and the executive.

Properly designed ES modules allow the executive and the applications programs to be developed independently from each other thus setting the stage for software modularity.

Some of the subfunctions included the executive service function are:

TASK SERVICE

SCHEDULE
CANCEL
TERMINATE

DATA SERVICE

READ
WRITE

EVENT SERVICE

SIGNAL
EVENT WAIT

TIMING SERVICE

TIME WAIT

Data Control (DC)

The data control function does the data management function for a processing element (e.g., controls global data storage pools within memory allowing updates only on a noninterference basis). Data control also maintains queues for the bus controller function for actual bus transmission. Additionally, data control sets BIU for actual bus transmissions by setting up tables for the BIU to utilize.

Local Synchronization (LS)

This module is called by local control (LC) upon receipt of a system synchronization signal and it handles any processing associated with system synchronization. This processing includes interfacing with data control (DC) to set up any data base or hardware registers associated with synchronized transmission. Also, the LS interfaces with executive service (ES) to handle any events associated with system synchronization.

Local Error Handling (LE)

This module is system dependent and is restricted to handling errors associated with the local I/O devices and processor transient errors.

Local I/O (LI)

This module handles the hardware related tasks associated with the reading and writing of data to any nonbus devices interfaced with the host processor. Possible devices include sensors, actuators, CRTs, magnetic tape transports or online disk/drum type mass memories. The LI module would be activated by the data control function.

Processor Interface (PI)

This module handles the software/hardware interface with the host processing element, including interrupt handling, timer control and the hardware register selection.

There are other functions that are really part of an executive computer program that have not been discussed. These include such functions as initialization, reload and the loader. These functions do not operate in realtime and typically are not architecture dependent. Therefore, these functions need not be included in this discussion.

MODULAR EXECUTIVE CONFIGURATIONS

Utilizing the functional modules previously defined, plus rigidly defined interfaces (illustrated in figure 5 and table 1), executive configurations can be assembled to satisfy the processing requirements of future distributed architectures. These configurations are many, but there is an inherent amount of commonality between the architectures and the number of combinations is limited. Starting with a single level network with a single point stationary bus control, there are two executive configurations that can be used: the master (bus controller) and the remote (non bus controller). These two configurations are illustrated (using the modules defined in figure 5) in figures 6 and 7. The same modules can be used to support the operational requirements of a hierarchical executive given that the same defined interfaces are used and there is sufficient hardware capability available.

It is believed that a significant number of the executive modules described above will be unchanged regardless of configuration or architecture. To illustrate this point, five executive configurations are described below.

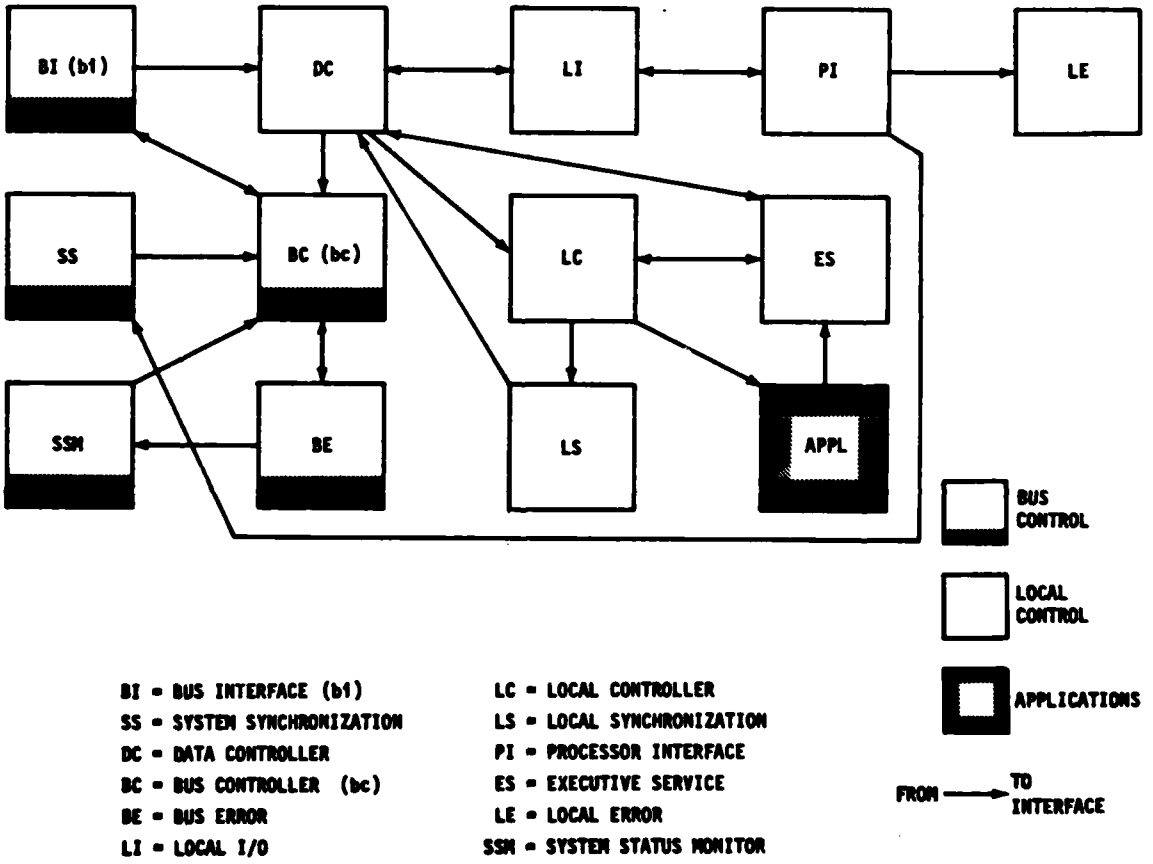


FIGURE 8 PROPOSED EXECUTIVE MODULE INTERFACE SCHEMATIC

MODULE	MASTER/REMOTE	EXECUTIVE FUNCTION	INTERFACE
BC	MASTER	BUS CONTROL	BI, BE
BI	MASTER	BUS CONTROL	BC, DC
bc	REMOTE	BUS CONTROL	bi
bi	REMOTE	BUS CONTROL	bc, DC
SS	MASTER	BUS CONTROL	BC
SSM	MASTER	BUS CONTROL	BC
BE	MASTER	BUS CONTROL	BC, SSM
DC	n/a	LOCAL CONTROL	BC, LC, LI, ES
LC	n/a	LOCAL CONTROL	ES, LS
LI	n/a	LOCAL CONTROL	DC, PI
LS	n/a	LOCAL CONTROL	DC,ES
ES	n/a	LOCAL CONTROL	DC, LC
PI	n/a	LOCAL CONTROL	LI, LE, SS*
LE	n/a	LOCAL CONTROL	

* MASTER ONLY

TABLE 1. EXECUTIVE FUNCTIONAL MODULE SUMMARY

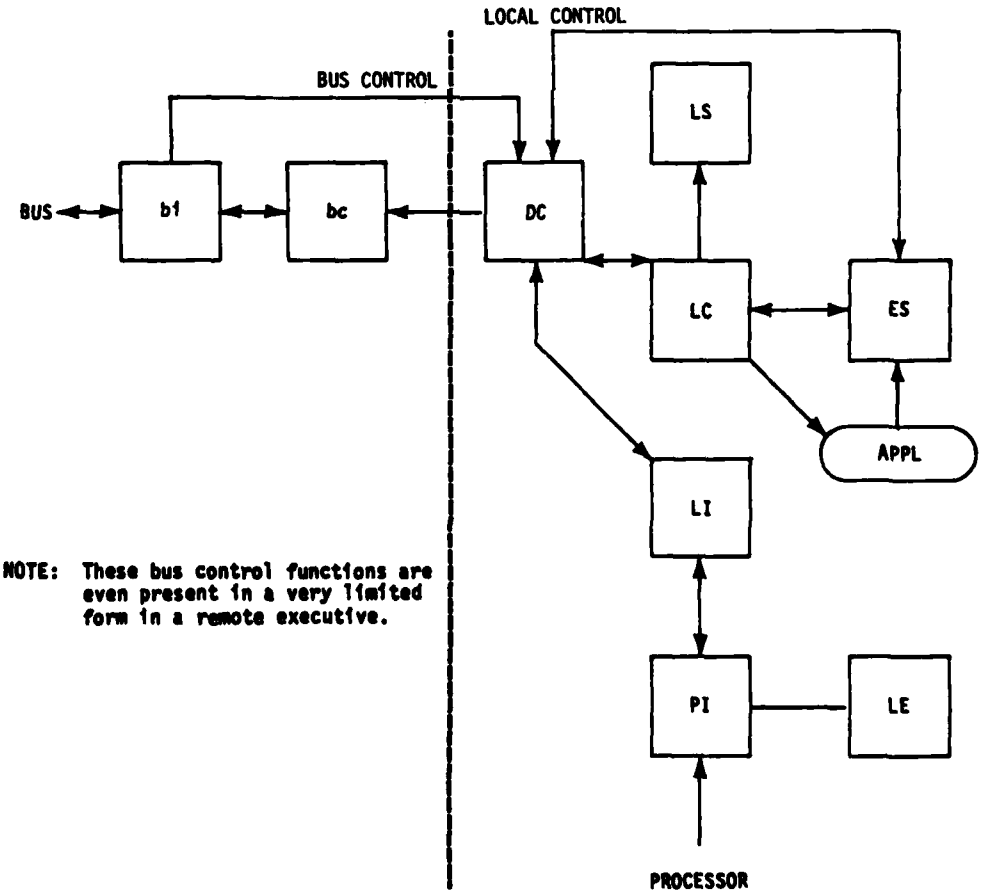


FIGURE 6 REMOTE EXECUTIVE (NON BUS CONTROLLER)

STATIONARY SINGLE-POINT BUS CONTROL EXECUTIVE

The software configuration for this type of executive computer program is illustrated in figure 7. The functional description of these modules was presented earlier.

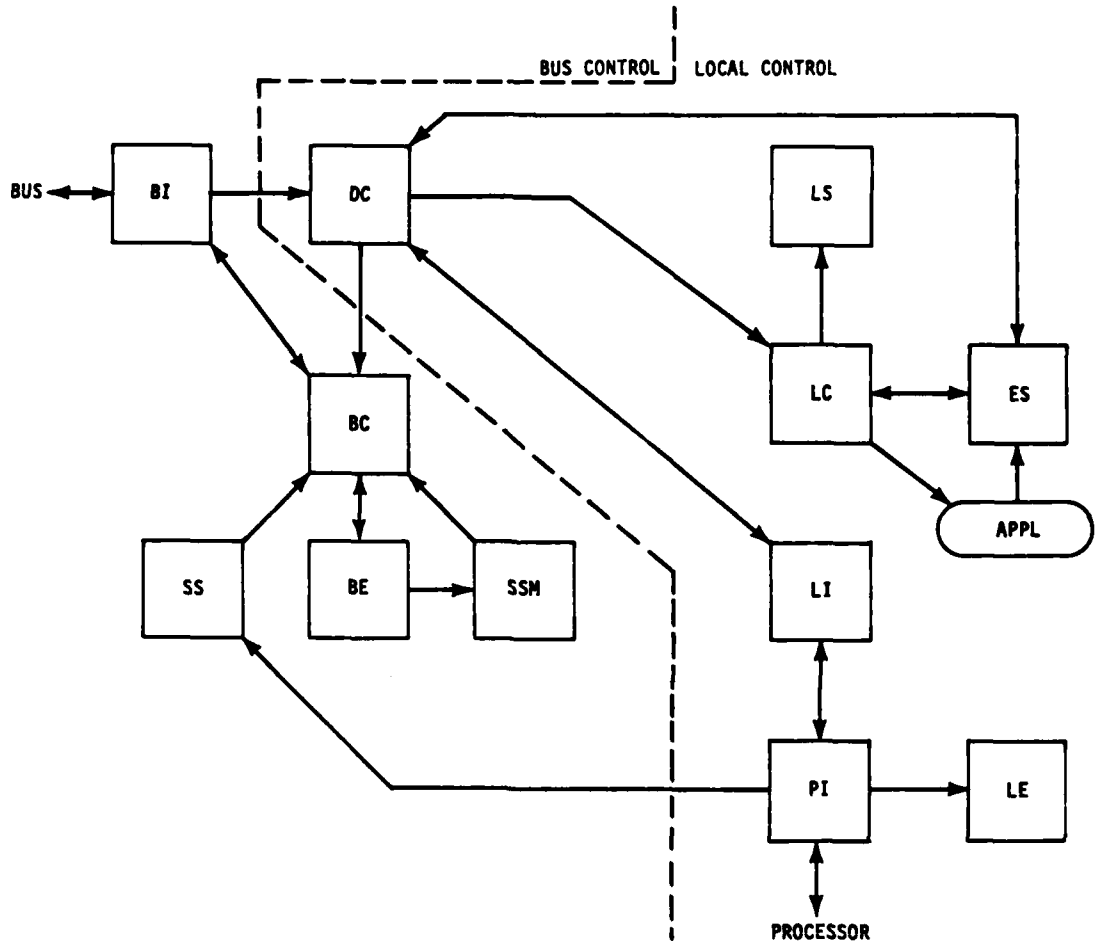


FIGURE 7 MASTER EXECUTIVE

There are two key elements in this configuration which differ from the others that will be discussed. The first is the function of bus acquisition and bus handover. Bus acquisition occurs at initialization time and bus handover does not occur (stationary single-point bus control). The second is the bus error handling function. With the stationary single-point bus control executive, the error handling function is global and necessarily complex to account for all of the devices in the network.

If there are any other processors in a stationary single-point bus control system, there will be another executive configuration which is known as a "remote executive" (this is true with any architecture where there are processors in the network which are never bus controllers, but it is predominate with the stationary single-point bus control mechanism). Using the modules which were described earlier and discussed relative to being used in the stationary single-point bus control executive, a remote executive can be configured. Again, note that the interfaces between the modules must be rigidly enforced for the modular executive scheme to be effected.

Examination of figure 6 reveals that the functional modules SS, BE and SSM are not present in the configuration. Additionally, functional modules BI and BC are replaced by bi and bc respectively because these functions, however slight, are still present in a remote executive. Module PI ignores the timer interrupt to generate the system

synchronization signal. Aside from these particular differences, the remaining functional modules are the very same functional modules which were utilized in the stationary single-point bus control (master) executive.

NONSTATIONARY BUS CONTROL EXECUTIVE

The three "types" of nonstationary bus control executives are the polling, the round-robin and the contention bus control transfer schemes. The executive differs in each case by the method of transferring control throughout the network. Again, using the same set of modules as was used to configure the stationary single-point bus control executive, the executive programs to satisfy these three bus control transfer schemes will be examined.

Polling

Examining figure 7, all of the modules with the possible exception of SS and SSM would be used to configure the executive for this bus control mechanism. The reason that these functional modules may not be used in all executives is that with nonstationary control it is doubtful that each potential bus controller will perform system synchronization and system status monitoring. These functions probably would be accomplished in one processing element. The capabilities of the BE functional module might be lessened by the same reasoning. The only module that needs to be changed is the BI module, because it possesses the capability to respond to a poll positively upon a request from the BC functional module to acquire bus control. This functional module also has the capability to start a poll and relinquish bus control when requested by the BC module.

The BC module differs slightly in that it ascertains whether or not the host processing element is the current bus controller. If it is the bus controller, then transmission is scheduled by the BC. If not, then the BC requests that BI acquire control of the bus and delay the transmission sequence until bus control is acquired.

All other executive modules could be used unchanged as required to support mission requirements.

Round-Robin

The polling bus control executive discussion above generally applies to the round-robin nonstationary bus control executive, and all modules are essentially the same for the two configurations. There would be, however, a difference in the BI functional module since in a round-robin bus control transfer scheme there is no poll to pass control. Typically, this function is accomplished by a fixed address list used by the current controller to determine the next potential controller.

The BC functional module would work in the same manner as with the polling executive in that it determines whether the host processing element is or is not the current bus controller. If not, the BC interfaces with BI to request and acquire bus control.

The BE functional module has to be more sophisticated with this bus control transfer scheme, because the loss of a processing element could result in the loss of bus control transfer ability.

Contention

The discussions above also apply to this bus control transfer scheme in that the only real change to a functional module would be isolated to the BI functional module where the changes to support the contention bus acquisition and handover would be implemented. As with polling and round-robin, the remainder of the executive functional modules would be unchanged. Table 2 summarizes this discussion of executive software functional module usage per configuration.

	EXECUTIVE TYPE				
	STATIONARY		NONSTATIONARY		
	MASTER	REMOTE	POLLING	ROUND-ROBIN	CONTENTION
BI	-	n/a (bi)	X	X	X
BC	-	n/a (bc)	-	-	-
BE	-	n/a	+	+	+
SS	-	n/a	+	+	+
SSM	-	n/a	+	+	+
DC	-	-	-	-	-
LC	-	-	-	-	-
LS	-	-	-	-	-
BS	-	-	-	-	-
PI	-	-	-	-	-
LE	-	-	-	-	-
LI	-	-	-	-	-

LEGEND:
X = Change necessary to support control scheme
- = No change necessary to support control scheme
+ = Potential change necessary to support control scheme

TABLE 2 SUMMARY OF EXECUTIVE FUNCTIONAL MODULE USAGE

4.0 HIERARCHICAL NETWORK CONSIDERATIONS

Regardless of the bus control mechanism employed in future avionics systems, it appears the hierarchical networks will definitely be widely used in the future (3). In some cases the hierarchical network may be controlled by different bus control mechanisms. An example would be polling nonstationary bus control on the global bus and single-point stationary on a subbus or buses.

There are a number of reasons to justify the hierarchical network and subsequent hybrid architecture. To begin with, this type of architecture offers the greatest flexibility and growth potential and at the same time probably offers the greatest reliability. Reliability is enhanced because fault/failure detection, isolation and handling can be accomplished at the subsystem level where the fault/failure actually occurs. This in turn simplifies the system wide Error Handling and Recovery EHAR function because an all encompassing system wide EHAR function is no longer required. Flexibility is enhanced because the subsystem, however complex, will continue to interface to the global bus at a data level allowing the system relative independence from knowledge of the components of the subsystem. Another key point is controlability, which is always an issue in distributed systems, is actually simplified with the hierarchical architecture due to the functional isolation between subsystems. The system controller "knows" the subsystem as a single functional unit regardless of the number of actual components (4).

The degree of sophistication at the subsystem level is of concern primarily to the subsystem designer, not to the system level designers. Basically there are two approaches to connecting major subsystems. One method, as discussed by Scarpino and Weber (3), simply connects the subsystem elements to a processor that connects to the global bus. The other, discussed by Edwards and Hubens (4) and Dennison and Dewey (5) actually distributes the subsystem elements on a local data bus similar to the global data bus.

An example of a hierarchical (interbus) executive configuration is illustrated in figure 8. This particular configuration assumes that the executive will perform as a remote on the global bus and as a master to the local lower bus or sub bus.

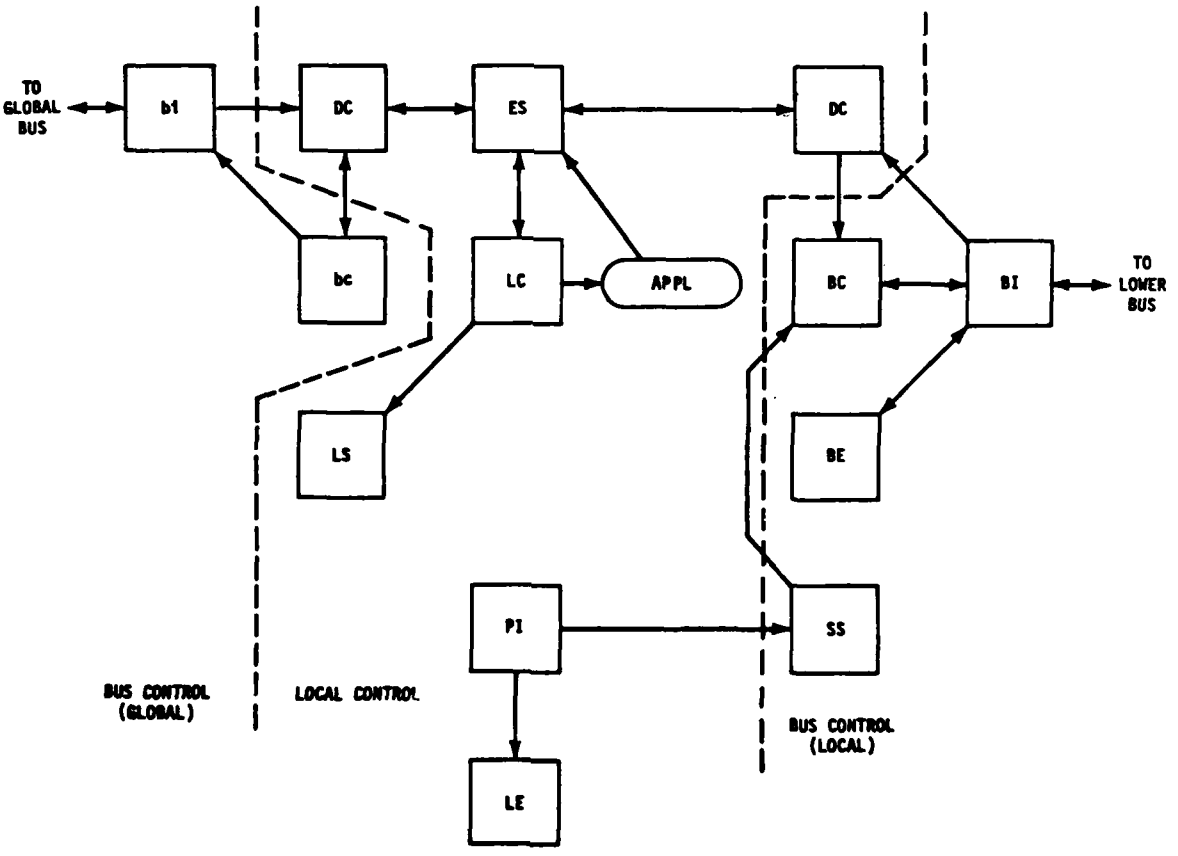


FIGURE 8 HIERARCHICAL EXECUTIVE CONFIGURATION

The functional modules which make up this configuration are basically the same as were presented earlier. The primary difference is the ES module which must interface with two DC modules, one module for each bus. This module is necessary to prevent the problem of simultaneous update if both buses were allowed to write into common memory. In order to satisfy these requirements, the ES module must be signalled whenever data arrives from either bus. When data arrives there are two possible courses of action. One, the data is to be processed in this processor or the data is to be sent to the other bus unprocessed. In the second case, the ES module moves the data from an input buffer on one bus to the output buffer for the other bus, and then signals the DC that there is data to be written. The other functional modules are virtually the same as in the configurations discussed earlier, providing the interfaces are properly maintained.

Of course, there are other issues that must be considered, such as responding to hardware interrupts. Interrupts can be handled through an interrupt priority scheme that allows processing of the highest level interrupts first. There would also be three basic levels of software priority which would have to be maintained for bus control for each bus (global and bus) and for local control within the processing element itself.

5.0 CONCLUSIONS

The advent of the microprocessor and the rapid maturation of this hardware is going to offer tremendous capability to avionics processing. Because of this growth, there is considerable concern about the impact upon operational software both existing (retrofit or upgrade) and planned. Obviously, there will be an impact due to the use of new hardware as there always has been and probably will be in the future. However, one thing remains clear; the basic functions of an executive computer program remain basically the same from airframe to airframe and from mission to mission. Only a small portion of an executive program actually changes and those portions of the executive that do not change should be reused. Those that do change can be isolated (functionally and modularily) so that engineering resources can be utilized to solve the new problem.

With a good definition of functional modularity and interface definition much like the USAF DAIS program has started, the impact on avionics software can be minimized.

REFERENCES

1. L. A. Smith, W. A. Crossgrove, et. al, Seattle, Washington, USA, The Boeing Company, Advanced Avionics Systems for Multimission Applications, Interim Technical Report, No. 1, (AF-TR-77-1252), 1978, Volume II, Appendix B, page 1.

2. ibid, Appendix C.

3. F. Scarpino, Maj. J. Weber, USAF, A DAIS Architecture Utilizing Microprocessors, NAECON '76 Record, pages 82 - 88.

4. Dr. J. Edwards, F. Hubans, General Dynamics, A Hierarchical Network for Avionics Systems, NAECON '78 Record, pages 129 - 138.

5. D. E. Dewey, R. G. Dennison, The Boeing Company, System Design Considerations for Microprocessor Based Distributed Architectures, MEDE '77 Conference Proceedings, pages 99 - 118.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-AG-258	ISBN 92-835-0267-1	UNCLASSIFIED

5. Originator Advisory Group for Aerospace Research and Development
North Atlantic Treaty Organization
7 rue Ancelle, 92200 Neuilly sur Seine, France

6. Title
GUIDANCE AND CONTROL SOFTWARE

7. Presented at

8. Author(s)/Editor(s)		9. Date
Various	Edited by Louis J. Urban	May 1980
10. Author's/Editor's Address		**11. Pages**
Various	Technical Director ASD/AX Wright-Patterson AFB OH 45433, USA	230

12. Distribution Statement This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.

13. Keywords/Descriptors

Software design and management
- Verification
- Validation
- Maintenance management

Software application
(IRAS satellite, F.16, Tornado, Concorde, Sea Harrier, Helicopter, Space Shuttle P.3.C)

14. Abstract

The development of Computer Programs, which are referred to as Software is currently on the critical path of all weapon systems and developments. The AGARDograph, prepared at the request of the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development. The AGARDograph is organized into two major parts: Part I deals with software design and management while Part II covers software applications.

AGARDograph No.258
Advisory Group for Aerospace Research and Development, NATO
GUIDANCE AND CONTROL SOFTWARE
Edited by L.J.Urban
Published May 1980
230 pages

The development of Computer Programs, which are referred to as Software is currently on the critical path of all weapon systems and developments. The AGARDograph, prepared at the request of the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development. The AGARDograph is organized into two major parts: Part I deals with software design and management while Part II covers software applications.

ISBN 92-835-0267-1

AGARD-AG-258

Software design and management
- Verification
- Validation
- Maintenance management

Software applications (IRAS satellite, F.16, Tornado, Concorde, Sea Harrier, Helicopter, Space Shuttle P.3.C)

AGARDograph No.258
Advisory Group for Aerospace Research and Development, NATO
GUIDANCE AND CONTROL SOFTWARE
Edited by L.J.Urban
Published May 1980
230 pages

The development of Computer Programs, which are referred to as Software is currently on the critical path of all weapon systems and developments. The AGARDograph, prepared at the request of the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development. The AGARDograph is organized into two major parts: Part I deals with software design and management while Part II covers software applications.

ISBN 92-835-0267-1

AGARD-AG-258

Software design and management
- Verification
- Validation
- Maintenance management

Software applications (IRAS satellite, F.16, Tornado, Concorde, Sea Harrier, Helicopter, Space Shuttle P.3.C)

AGARDograph No.258
Advisory Group for Aerospace Research and Development, NATO
GUIDANCE AND CONTROL SOFTWARE
Edited by L.J.Urban
Published May 1980
230 pages

The development of Computer Programs, which are referred to as Software is currently on the critical path of all weapon systems and developments. The AGARDograph, prepared at the request of the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development. The AGARDograph is organized into two major parts: Part I deals with software design and management while Part II covers software applications.

ISBN 92-835-0267-1

AGARD-AG-258

Software design and management
- Verification
- Validation
- Maintenance management

Software applications (IRAS satellite, F.16, Tornado, Concorde, Sea Harrier, Helicopter, Space Shuttle P.3.C)

AGARDograph No.258
Advisory Group for Aerospace Research and Development, NATO
GUIDANCE AND CONTROL SOFTWARE
Edited by L.J.Urban
Published May 1980
230 pages

The development of Computer Programs, which are referred to as Software is currently on the critical path of all weapon systems and developments. The AGARDograph, prepared at the request of the Guidance and Control Panel of AGARD, brings together related experience in the NATO community as a guide for future guidance and control software development. The AGARDograph is organized into two major parts: Part I deals with software design and management while Part II covers software applications.

ISBN 92-835-0267-1

AGARD-AG-258

Software design and management
- Verification
- Validation
- Maintenance management

Software applications (IRAS satellite, F.16, Tornado, Concorde, Sea Harrier, Helicopter, Space Shuttle P.3.C)